nomodo - Manage your system by yourself

Autori (in ordine alfabetico): Giuseppe Glorioso Lucia Polizzi

June 30, 2018

Contents

1 Introduzione

1.1 Introduzione al progetto

Il progetto nomodo nasce dalla necessitá di un applicativo di gestione dei sistemi Ubuntu che sia più immediato ed accessibile rispetto al classico terminale, e quindi utilizzabile anche dagli utenti che per un motivo o per un altro non possono o non vogliono avere a che fare con il terminale. Nomodo si prende in carico di eseguire tutte le chiamate al terminale o meno per eseguire operazioni atte alla gestione del sistema presentando all'utente una interfaccia web chiara e comprensibile. Per operazioni in questo caso si intendono l'aggiornamento, la manutenzione e il miglioramento del sistema come ad esempio l'installazione dei pacchetti, la ricerca e la modifica dei file, così come operazioni di più alto livello come la gestione basilare del web server Apache.

1.2 Informazioni tecniche

Python + Flask L'applicativo scritto in python è basato sul framework Flask, utilizzato tra l'altro come webserver per l'accesso al pannello. Durante la fase di sviluppo si è utilizzato nginx come reverse proxy in modo da poter raggiungere il pannello web sulla porta 80 e non sulla 5000. È stata presa poi in seguito la decisione di lasciare che l'applicativo girasse sulla porta 5000 in quannto meno comune e quindi meno alla mercé degli hacker.

L'applicazione è stata quindi divisa in modo netto nelle due componenti fondamentali, il Frontend e il Backend che anche andremo quindi ad analizzare qui brevemente e più approfinditamente nei capitoli successivi:

- Il **backend** consiste in una serie di funzioni raccolte in una serie di file a mò di libreria, risiedenti nella cartella `systemcalls` (come ad es. `system.py` o `user.py`), utilizzati sia per la raccolta di dati sia per eseguire azioni sul sistema che non necessitano di output in uscita
- Il **frontend** rappresenta la parte grafica dell'applicativo web, e utilizza le funzioni del backend per la ricerca di informazioni e per la modifica alle componenti del sistema inclusa la modifica dei file quali i file di configurazioni

MongoDB Ogni operazione sensibile effettuata tramite l'applicazione comporta la memorizzazione delle modifiche che comporta la stessa in documento di mongodb, cosí da poter risalire alla storia delle operazioni effettuate e tentare un revert delle modifiche in caso ad esempio il sistema perda di stabilità o le modifiche non portino al risultato sperato. Tali operazioni possono riguardare ad esempio la modifica di un file, o la rimozione di un pacchetto dal sistema. Ogni log in mongodb presenta inoltre un flag **status** che indica se l'operazione eseguita sia andata a buon fine o meno, in modo da rendere più chiara la navigazione tra i log e dare la possibiltà all'utente di filtrarli in base a questo campo. [1]

2 Installazione

Questa sezione spiega brevemente come effettuare l'installazione del progetto `nomodo`.

[1] Le uniche operazioni memorizzate nel database sono quelle relative all'utilizzo dell'applicativo; una modifica effettuata direttamente sul sistema ad esempio tramite il terminale va incontro alle regole del sistema Ubuntu e ogni modifica potrebbe essere irreversibile. In questi casi fare riferimento ai log del sistema che è possibile trovare al percorso `/var/log/` o sul pannello web alla sezione `Log`.

1. Assicurarsi che ci si trovi su un sistema Ubuntu lanciando il comando `cat /etc/os-release`. Se il file non esiste o non contiene la stringa "Ubuntu" allora il sistema non è adatto all'installazione di nomodo (per adesso)
2. Assicurarsi che il sistema sia connesso ad internet
3. Spostarsi nella cartella *install* del progetto e lanciare l'eseguibile *install.sh*. Lo script provvederà all'installazione e chiederà ad un certo punto la password per l'utente mongo. Inserire una password a propria scelta
4. Se non si sono verificati errori l'esecuzione è terminata e nomodo è stato installato sul pc

3 Backend

Come anticipato in sezione 1.2 il backend è composto da una serie di funzioni raggruppate per categoria che fanno utilizzo di varie librerie python per compiere operazioni che possono o meno alterare lo stato del sistema. Allo stato attuale le categorie che compongono il backend sono le seguenti:

- Utenti
- Network
- Cron
- Sistema
- Apache
- Database
- File
- Logs

Interfaccia al frontend Ogni funzione chiamata restituisce sempre un dizionario contenente almeno n codice di ritorno e il logid del documento inserito in mongo con un mongo `_id` se applicabile[2] oppure un logid `None` se non è stato creato alcun log. Distinguiamo quindi 2 casi in base al valore della variabile `returncode`:

- Se $returncode = 0$ l'operazione è andata a buon fine e il dizionario conterrà una terza variabile `data` che conterrà i dati richiesti se la funzione chiamata è tesa per restutuire output oppure sarà una variabile nulla se la funzione non restituisce output
- Se $returncode \neq 0$ c'è stato un errore durante l'esecuzione dell'applicazione e il dizionario restituito conterrà quindi una terza variabile `stderr` il cui valore è un messaggio di errore e, se l'errore è dato da un comando eseguito in bash, il comando che una volta lanciato ha generato l'eccezione.

Il frontend o l'utente che voglia chiamare per qualsivoglia motivo le funzioni del backend direttamente, potrá farlo quindi nel seguente modo::

[2] Cioè in caso l'operazione sia una operazione sensibile e richieda quindi un inserimento in mongo per tenere traccia della stessa

```
data = getifacestat()
if data['returncode'] is 0:
        data = data['data']
else:
        print( data['stderr'] )

pprint(data)
```

Si fa notare inoltre che il formato dell'output delle funzioni del backend è stato adattato agli scopi del frontend e mai per essere chiaro all'utente che voglia lanciare queste funzioni direttamente.

subprocess Le funzionalità di Python più utilizzata per la realizzazione dell'applicazione sono senza dubbio quelle appartenenti alla libreria subprocess, che permette di eseguire comandi come se si stessero eseguendo in bash. Si è cercato il più possibile di limitare l'utilizzo di questa libreria ma le sue funzionalità si sono rese necessarie nella maggior parte dei casi delle funzioni del backend, a causa dello scarsa agilità che ha python di interfacciarsi col sistema sottostante. In generale l'esecuzione di un comando avviene nel seguente modo:

```
command = ['ifconfig', '-a']
try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True)
except CalledProcessError as e:
        return command_error(e, command, logid)

return command_success( output )
```

Tutte le funzioni di nomodo ritornano o con un `command_success` in caso l'operazione sia andata a buon fine o con un `command_error` in caso il comando non vada a buon fine e venga lanciata l'eccezione `CalledProcessError`. In entrambi i casi viene restituito il dizionario menzionato in sezione 2. Un esempio di comando che non restituisce output è il seguente:

```
def removeuser(user, removehome=None):

    logid = mongolog( locals(), getuser(user) )


    try:
        command = ['deluser', user]
        if removehome: command.append('--remove-home')

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success(logid)
```

Nelle prossime sezioni verranno analizzate tutte le categorie e spiegato il funzionamento di ogni funzione che contengono.

3.1 Utilities

Questa categoria contiene per la maggior parte funzioni che non vengono mai richiamate direttamente dal frontend, ma vengono utilizzate dalle altre funzioni del backend. Fanno eccezione le funzioni per il check e lo start del demone di mongo e quelle inerenti le opoerazioni sui file come as esempio writefile. Analizziamo le funzioni di questa libreria nelle prossime sezioni.

3.1.1 mongocheck()

```
def mongocheck():

    try:
        command = ['systemctl', 'status', 'mongod', '-q', '-n0', '--no-
            pager']
        check_call(command)
    except CalledProcessError:
        stderr='Mongo daemon is not running. Mongo is a requirement to
            use nomodo. To achieve this please launch this command on
            terminal: systemctl start mongod'
        return command_error( returncode=42, stderr=stderr )

    return command_success()
```

La funzione nasce per essere chiamata direttamente dal frontend e verifica se il demone di mongod è attivo, restituendo errore in caso contrario. Il fontend una volta chiamata questa funzione blocca completamente l'utilizzo di nomodo da parte dell'utente se questa restituuisce un returncode diverso da 0, ad indicare che mongo non è stato avviato. Propone quindi all'utente di attivare il demone attraverso un bottone che chiama la funzione mongostart().

Parametri La funzione non prende parametri.

Funzionamento Semplicemente costruisce ed esegue il comando di systemd `systemctl` che chiamato come incodice controllo lo stato attuale del demone di mongo, ossia `mongod`. Il comando viene chiamato col parametro `--no-pager` in modo da non bloccare lo stdin e quindi l'intera applicazione. Se mongo non è attivo viene restituito un codice di ritorno diverso da 0 e entrando nell'`exception` viene restituito all'utente un messaggio di errire.

Return Se l'operazione va a buon fine restituisce un dizionario di successo vuoto dove il `returncode` è 0. Altrimenti restituisce il dizionario di errore con il returncode e il messaggio di errore che si vede in codice.

3.1.2 mongostart()

```
def mongostart():

    try:
        command = ['systemctl', 'start', 'mongod']
        check_call(command)
    except CalledProcessError as e:
```

```
        return command_error( e, command )

    return command_success()
```

È incaricata di far partire il demone si mongo (`mongod`) e lavora insieme alla funzione mongocheck().

Parametri Non prende alcun parametro.

Funzionamento Semplicemente costruisce il comando per lanciare il demone usando `systemctl` e lo lancia controllando che non si verifichino eccezioni.

Return In caso il demone di mongo venga avviato senza problemi restituisce il dizionario di successo col solo codice di ritorno impotstato a 0, in caso contrario il dizionario di errore contenente codice di ritorno e messaggio di errore generati dal sistema.

3.1.3 mongolog()

```
def mongolog(params, *args):

    dblog = dict({
        'date': datetime.datetime.utcnow(),      #Operation date
        'funname': inspect.stack()[1][3],        #Function name
        'parameters': params,                    #Called function's
           parameters
    })

    for arg in args:
        dblog.update( arg )

    #ObjectID in mongodb
    return db.log.insert_one( dblog ).inserted_id
```

Viene chiamata ogni volta che una funzione sia classificata come **sensibile** cioè che va a modificare lievemente o pesantemente il sistema e prende in carico di creare un log mongodb contenente le operazioni eseguite e i dati modificati dalla funzione. Accetta N parametri di cui il primo (obbligatorio) è la lista di parametri con cui è stata lanciata la funzione di cui si sta memorizzando il log. Ad esempio in

```
def ifacedown( iface ):
        logid = mongolog( locals() )
        ...
```

il primo parametro è `locals()` che contiene la variabile `iface` che verrà quindi memorizzata nel log di mongo; il secondo parametro (opzionale) può essere uno o più dizionari da unire al dizionario memorizzato in mongodb. Ad esempio nella funzionei `addusertogroups`:

```
def addusertogroups(user, *groups):

    #Logging operation to mongo first
    userinfo = getuser(user)
```

```
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
    ...
```

Si è deciso che prima di aggiungere un utente a dei nuovi gruppi si va a memorizzare in mongo non solo `locals()` e quindi `user` e `*groups` ma anche le informazioni sull'utente ricavate attraverso la funzione `getuser()` e passate a `mongolog()` come secondo parametro.

Il dizionario di base memorizzato in mongo è formato da tre elementi:

- La data in cui viene effettuata l'operazione
- Il nome della funzione che ha chiamato `mongolog`, ricavata tramite il supporto della libreria `inspect`
- I parametri della funzione che chiama, come spiegato in precedenza, e ottenuti chiamando la funzione `locals()`

3.1.4 mongologstatus() e funzioni collegate

```
def mongologstatus(logid, status):

    return db.log.update_one(
        {'_id': logid},
        {'$set': { 'status' : status }},
        upsert=False
        )

def mongologstatuserr(logid, status='error'):
    return mongologstatus(logid, status)
def mongologstatussuc(logid, status='success'):
    return mongologstatus(logid, status)
```

Parametri Accetta 2 parametri:

- `logid`: è il logid del documento di mongo a cui aggiungere o modificare il campo status
- `status='error'`: è lo stato da assegnare la log individuato da `logid`

Questa funzione è intesa per aggiungere o modificare il campo `status` di un log di MongoDB. Le funzioni di nomodo (come è giusto che sia) creano un documento di mongo per memorizzare le informazioni sull'operazione prima di procedere all'operazione stessa. In caso un'operazione non andasse nel modo aspettato bisognerebbe quindi marcare il documento appena creato in mongo in modo da avvisare l'utente che sta consultando il log che l'operazione riferita a quel documento non è andata a buon fine. La memorizzazione del log avviene quindi nei seguenti step:

1. Viene lanciata la funzione che richiede la memorizzazione del log e quindi mongolog(), che va a creare il log senza nessuna indicazione sul successo o meno dell'operazione
2. Dopo l'esecuzione della funzione viene chiamata `command_success` se l'operazione è andata a buon fine; la prima operazione che questa va ad eseguire è chiamare a sua volta la funzione `mongologstatussuc()` che chiama `mongologstatus()` con il parametro `status='error'` aggiungendo tale campo `status` al log di mongo ed indicando la buona riuscita dell'applicazione all'utente che andrà ad analizzare i log
3. In caso invece la funzione vada in errore viene chiamata command_success che chiama `mongologstatuserr()` che chiama `mongologstatus()` con il secondo parametro `status='error'` aggiungendo tale campo al log di mongo
4. In caso invece si voglia personalizzare il campo `status` basta quindi che la funzione chiami direttamente `mongologstatus()` con il secondo parametro `status` ad un qualsivoglia valore si voglia inserire, ad es. `status='canceled'`

Si intuisce quindi da questi step che un log che non abbia il campo status indica un crash della funzione nel codice che è intercorso tra la memorizzazione del log e l'aggiunta del campo `status`.

L'operazione deve fallire se il documento indicato da `logid` non esiste, quindi si è aggiunta la direttiva `upsert=False`.

Ecco un esempio che mostra lo stato di un log appena aggiunto (senza il campo `status`) e al termine dopo aver chiamato `command_success`:

```
> db.log.find()
{ "_id" : ObjectId("5ae596d4bf3bd205c1aeea25"), "parameters" : { "shell"
   : "/bin/bash", "user" : "giuseppe2", "password" : "test" }, "funname"
   : "adduser", "date" : ISODate("2018-04-29T09:56:36.627Z") }
> db.log.find()
{ "_id" : ObjectId("5ae596d4bf3bd205c1aeea25"), "parameters" : { "shell"
   : "/bin/bash", "user" : "giuseppe2", "password" : "test" }, "funname"
   : "adduser", "date" : ISODate("2018-04-29T09:56:36.627Z"), "status" :
   "success" }
```

Return Restituisce l'object id del documento mongo di cui ha aggiornato lo stato.

3.1.5 command_success

```
def command_success( data=None, logid=None, returncode=0 ):

    if logid:
        mongologstatussuc( logid )

    return dict({
        'returncode': returncode,
        'data': data,
        'logid': logid
    })
```

La funzione `command_success` fondamentalmente costruisce il dizionario da restituire all'utente quando una funzione del backend ha finito le sue opoerazioni e non ci sono stati errori durante l'esecuzione. Insieme alla sorella command_error sono le uniche due funzioni chiamate al termine di una funzione del backend.

Parametri Accetta tre parametri:

- `data=None`: Sono i dati da restituire all'utente se la funzione che l'ha chiamata li genera. Di base è `None`
- `logid`: Il logid a cui aggiungere il campo `status` e da restituire all'utente nel dizionario come campo del dizionario. Di base è `None` in quanto il chiamante potrebbe non aver generato un mongolog per l'operazione che ha effettuato
- `returncode`: È il codice di ritorno che verrà inserito nel dizionario. Essendo questa funzione invocata ogni qualvolta il chiamante esegue tutte le operazioni senza errore di base questo parametro è 0 ad indicare successo e può quindi essere omesso, ma può essere personalizzato passandolo alla chiamata

Funzionamento La prima operazione eseguita è la chiamata a`mongologstatussuc()` per aggiungere al log di mongo il campo `status`, questo solo in caso il parametro `logid` sia non nullo e quindi la funzione chiamante ha dovuto memorizzare un mongolog. Successivamente va a costruire il dizionario da restituire formato dal codice di ritorno, i dati voluti dall'utente (se disponibili, altrimenti `None`), e il *logid* dell'operazione.

Return Restituisce un dizionario che chiamiamo **dizionario di successo** che indica che una operazione è andata a buon fine attraverso il codice di ritorno sempre a 0. Inserisce inoltre i parametri passati come argomento all'interno del dizionario restituito. Da utilizzare come spiegato in sezione 2.

3.1.6 command_error

```
def command_error( e=None, command=[], logid=None, returncode=1, stderr='
    No messages defined for this error' ):

    if logid:
        mongologstatuserr( logid )

    return dict({
        'returncode': e.returncode if e else returncode,
        'command': ' '.join(command),
        'stderr': e.stderr if e else stderr,
        'logid': logid
    })
```

`command_error` è l'opposto di `command_success`. Come visto in sezione 2 viene invocato quando un comando lanciato attraverso la libreria `subprocess` fallisce nell'esecuzione. Può essere però anche usato per generare un dizionario di errore personalizzato.

Parametri Accetta 5 parametri:

- `e=None`: è l'oggetto creato nel caso in cui venga lanciata l'eccezione `CalledProcessError`
- `command=[]`: è il comando la cui esecuzione ha generato l'eccezione. Lista vuota di default
- `logid`: Il logid a cui aggiungere il campo `status` e da restituire all'utente nel dizionario come campo del dizionario. Di base è `None` in quanto il chiamante potrebbe non aver generato un mongolog per l'operazione che ha effettuato
- `returncode=1`: Un codice di ritorno personalizzato da inserire nel dizionario in caso il parametro `e` sia nullo
- `stderr='No messages defined for this error'`: È il messaggio di errore da inserire nel dizionario in caso il parametro `e` sia nullo

Funzionamento La funzione costruisce un dizionario da restituire all'utente contenente le varie informazioni sull'errore che è accaduto, ossia il codice di ritrorno, il messaggio di errore, il comando che ha generato l'errore (da passare in input) e il logid del documento in mongo che riguarda il comando/operazione.

Distinguiamo 3 casi:

- Viene generato un oggetto del tipo `CalledProcessError` appartenente a subprocess: in questo caso si passa alla funzione l'oggetto generato e il comando che ha causato l'errore. La funzione ricava automaticamente da questo oggetto il codice e il messaggio di erroree lo inserisce nel dizionario insieme al comando di `subprocess` che ha causato l'errore. È quindi in questo caso necessario passare almeno questo oggetto e il comando (che non è però strettamente necessario)
- Si vuole generare un errore personalizzato: in questo caso invece i parametri `e` e `command` non devono essere passati, e al loro posto vengono passati `returncode` e `stderr` che verranno inseriti nel dizionario da restituire.
- Si vuole generare un errore di default: in quest'ultimo caso basta chiamare la funzione senza passare alcun parametro e viene generato un errore di base in cui i valori del codice di ritorno saranno quelli assegnati di default e che si può vedere nella sezione *Parameters*

Oltre a restituire il dizionario di errore questa funzione, se il parametro `logid` è non nullo agisce come command_success aggiungendo quindi al log dell'operazione il campo `status` col valore `error`.

Return Resituisce un dizionario che chiamiamo **dizionario di errore** creato come descritto nel funzionamento e come si può vedere nel codice.

3.1.7 writefile

```
def writefile(filepath, newcontent=None, force=False):

    if not force:
        md5new = hashlib.md5()

        md5new.update( newcontent.encode() )
        md5new = md5new.hexdigest()
```

```
        md5old = hashlib.md5( open( filepath , 'rb' ).read() ).hexdigest()

        if md5new == md5old:
            return command_error( returncode=2, stderr='Nothing to write(
                no changes from original file). You can force writing
                using the parameter "force=True"' )

    localsvar = locals()
    del localsvar['newcontent']
    logid = mongolog( localsvar , filediff(filepath , newcontent) )

    opened = open(filepath , 'w')
    opened.write(newcontent)
    opened.close()

    return command_success( logid=logid )
```

Questa funzione è intesa per dare un supporto in caso l'utente voglia modificare un file. Implementa tutte le operazioni e i controlli che bisognerebbe effettuare prima della modifica di un file, incluso un mongolog da cui si possa risalire al contenuto del file prima della scrittura. È inoltre capace di restituire il contenuto di un file, utile in fase di pre-scrittura.

Parametri Sono 3 i paramentri accettati da questa funzione:

- `filename`: è il percorso del file sul sistema che si vuole modificare
- `towrite=None`: è il nuovo contenuto del file da scrivere sullo stesso
- `force=False`: questa variabile (di base a `False`) se `True` forza la scrittura del nuovo contenuto anche se questo non differisce dal contenuto originale del file

Funzionamento

```
    if not force:
        md5new = hashlib.md5()

        md5new.update( towrite.encode() )
        md5new = md5new.hexdigest()

        md5old = hashlib.md5( open( filename , 'rb' ).read() ).hexdigest()

        if md5new == md5old:
            return command_error( returncode=2, stderr='Nothing to write(
                no changes from original file). You can force writing
                using the parameter "force=True"' )
```

In questa seconda parte controlla la varibile `force`. Se questa è `false` genera l'md5 del nuovo contenuto (towrite) e del vecchio contenuto (quello del file) e nell'ultimo `if` ne controlla l'uguaglianza. Quindi se `force=False` e `md5new==md5old` non c'è necessità di scrivere il file e genera quindi un

messaggio di errore personalizzato (o potremmo dire di warning in questo caso) usando la funzione command_error, e che avverte il chiamante (o il fontend) che non è stata eseguita alcuna operazione ma che la si può forzare passando il parametro `force` col valore booleano `True`.

```
    localsvar = locals()
    del localsvar['towrite']
    logid = mongolog( localsvar, filediff(filename, towrite) )

    #Writing new content to "filename" file
    opened = open(filename, 'w')
    opened.write(towrite)
    opened.close()

    return command_success( logid=logid )
```

Si arriva quindi a questo pezzo di codice se `force=True` o se `force=False` e `md5new!=md5old`. Questa funzione andando a modifcare file che possono essere o meno importanti necessità di un log che memorizzi una quantità di informazioni tali da consentire il ripristino del file vecchio. Si intuisce però che memorizzare interamente il vecchio e il nuovo contenuto sarebbe impensabile per avere un mongolog decente e di piccole dimensioni. Si è quindi optato per inserire il solo diff tra vecchio e nuovo contenuto. In casi comuni e per i nostri scopi questa scelta porta ad una riduzione significativa della dimensione del log, in quanto raramemnte un utente cancella completamente e riscrive file di migliaia di righe di codice.
Passando al funzionamento quindi si memorizza dapprima ciò che restituisce la funzione `locals()` (che come spiegato restituisce i parametri con cui la funzione è stata chiamata) e si inserisce ciò che ritora nella variabile `localsvar`. Dopodichè da questa variabile (che è diventata un dizionario dopo l'assegnazione) si elimina il parametro `towrite` che a quanto ne sappiamo potrebbe anche essere di migliaia di righe. Si crea a questo punto un mongolog che non abbia più al suo interno il contenuto scritto ma ciò che ritorna la funzione filediff che come vedremo in seguito non fa altro che restituire il diff tra il vecchio e il nuovo contenuto del file. Una volta effettuato i controlli e memorizzati in mongo i dati necessari si procede finalmente a scrivere il nuovo contenuto sul file tramite le funzioni Python per la gestione dei file. Si invoca infine `command_success` passandogli il `logid`.

Return Restituisce il *dizionario di successo* generato dalla funzione `command_success` con all'interno il solo `logid` dell'operazione ad indicare la corretta scrittura del file. Il `returncode` di questo dizionario indica se la scrittura del file è andata a buon fine (=0) o meno (!=0). In caso questo returncode sia diverso di zero si ottiene il dizionario di errore con un breve messaggio con informazioni sullo stesso.

3.1.8 readfile

```
def readfile(filepath):

    if not os.path.isfile(filepath):
        return command_error( returncode=15, stderr="Specified file is
            not a file: "+filepath )

    try:
```

```
        with open(filepath, 'r') as content:
            return command_success( data=content.read() )
    except FileNotFoundError:
        return command_error( returncode=10, stderr='No file found on
           path : "'+filepath+'"' )
```

Utile a leggere il contenuto di un file.

Parametri Accetta un solo parametro che è il percorso del file sul sistema.

Funzionamento Verifica inizialmente che la stringa passata in input identifichi davvero un file (e non una directory ad esempio) usando la funzione `os.path.isfile()`. Poi semplicemente apre il file utilizzando la `with open`, ne legge il contenuto con la `read()` e ne restituisce quindi il contenuto nel dizionario di successo. Durante l'operazione viene controllato che non sia lanciata l'eccezione `FileNotFoundError` ad indicare che il file non esiste sulla macchina.

Return Restituisce il dizionario di successo contenente al campo `data` il contenuto del file indicato nell'unico parametro che accetta.

3.1.9 filediff

```
def filediff(filea, fileb):
    if not os.path.exists(filea):
        filecontent = filea
        filea = '/tmp/.nomodotempa'
        with open(filea, 'w') as opened:
            opened.write(filecontent)

    if not os.path.exists(fileb):
        filecontent = fileb
        fileb = '/tmp/.nomodotempb'
        with open(fileb, 'w') as opened:
            opened.write(filecontent)

    command = ['diff', filea, fileb]

    output = Popen(command, stdout=PIPE, universal_newlines=True).
       communicate()[0]

    return {'filediff': output }
```

La funzione `filediff` come si intusice dal nome serve a generare un diff tra 2 file. È usata principalmente dalla funzione writefile per le sue operazioni di log su mongo ma può anche essere chiamata direttamente dall'utente curioso o dal fontend.

Parametri Accetta 2 parametri. Questi non si differenziano l'uno dall'altro e possono essere un percorso ad un file sul sistema, o una stringa. È perfettamente legale che uno sia di un tipo e uno di un altro, ad es. `filea` può essere il percorso di un file mentre `fileb` una stringa che rappresenta il contenuto di un file, così come accade quando questa funzione viene chiamata da `writefile`.

Funzionamento Innanzitutto nei primi due `if` la funzione controlla se i parametri contengono una stringa che identifica il percorso di un file sul sistema o un contenuto usando la libreria `os`. Se uno dei due parametri non contiene un percorso si prende in carico di scrivere la stringa che contiene in un file temporaneo creato al momento. Ad es. se `filea` non è un file ma una stringa crea un nuovo file vuoto, `/tmp/.nomodotempa` in questo caso, e ci scrive il contenuto di `filea`.
Nella seconda parte viene composto e lanciato il comando `Popen` di `subprocess` [3] per effettuare il diff in linux style. Si è optato per tale diff in quanto nessuna funzione di Python restituisce il diff così chiaramente e in modo così adatto da essere inserito in un database.

Return Restituisce un dizionario con la sola chiave `filediff` dove il valore è il diff generato tra i due file o contenuti passatogli in input.

3.1.10 filedel

```
def filedel(filepath):

    logid = mongolog( locals() )

    if not os.path.isfile(filepath):
        return command_error( returncode=15, stderr="Specified file is
            not a file: "+filepath , logid=logid )

    try:
        os.remove( filepath )
    except FileNotFoundError:
        return command_error( returncode=10, stderr='File to remove not
            found: "'+filepath+'"', logid=logid )

    return command_success( logid=logid )
```

Nasce per eliminare un file dal sistema.

Parametri Accetta un solo parametro che è il percorso del file da rimuovere.

Funzionamento Inizialmente verficica che il file sia effettivamente un file usando la funzione `os.path.isfile()` per evitare di mandare il programma in errore. Poi semplicemente elimina il file utilizzando la funzione `remove` della libreria `os`. Durante la rimozione verifica che non si verifichi l'eccezione `FileNotFoundError`, lanciata se il file indicato dal percorso non esiste. In caso l'eccezione venga lanciata viene chiamata la funzione di errore con codice e messaggio di errore personalizzati.

Return Restituiusce il dizionario di errore costruito come si vede in codice se il file non esiste, il dizionario di successo altrimenti, contenente nella variabile `logid` l'object id del documento mongo creato e contenente le informazioni sull'operazione appena eseguita.

[3] Da notare che questa è una delle poche funzioni che usa `Popen` al posto di `check_output` e `check_call`. Questo è dovuto al fatto che i codici di ritorno del comando `diff` sono diversi da 0 se esistono differenze tra i 2 file analizzati, e utilizzare le due funzioni menzionate genererebbe un eccezione `CalledProcessError` che farebbe crashare il programma

3.1.11 filerename

```
def filerename(filepath, newname):

    logid = mongolog( locals() )

    if not os.path.isfile(filepath):
        return command_error( returncode=15, stderr="Specified file is
            not a file: "+filepath, logid=logid)

    try:
        newname = os.path.dirname(filepath) + '/' + newname
        os.rename( filepath, newname )
    except FileNotFoundError:
        return command_error( returncode=10, stderr='File to rename not
            found: "'+path+'"', logid=logid )

    return command_success( logid=logid )
```

La funzione nasce per rinominare un file.

Parametri Accetta 2 parametri:

- `filepath`: percorso+nomefile del file di cui cambiare il nome
- `newname`: il nuovo nome da dare al file. Non richiede che al suo interno sia presente nuovamente il percorso del file in quanto estratto presente nella variabile `filepath`

Funzionamento La funzione va dapprima a memorizzare un mongolog in quanto bisogna tenere traccia dell'operazione che è stata effettuata, e controlla se il file passato come argomento è effettivamente un file regolare. Si entra quindi in un `try` dove durante le operazioni viene verificato che non venga lanciata l'eccezione `FileNotFoundError`, lanciata se il file non esiste.
Il remane diretto con la funzione `os.rename()` richiede che il percorso del file sia presente anche nel nuovo nome dello stesso in quanto comportandosi come la funzione `mv` del terminale se non si specifica il percorso il file verrebbe spostato oltre che modificato. Si estrae quindi dalla variabile `filepath` il percorso originale del file che viene concatenato a `newname` nella chiamata a `os.rename` così da mantenere il percorso originale.

Return Restituisce il dizionario di successo col campo `logid` contenente l'object id del documento mongo contenente vecchio e nuovo nome file. Vedersi restituire questo dizionario vuole dire che l'operazione è andata a buon fine e che il file è stato rinominato con successo.

3.1.12 filecopy

```
def filecopy(src, dst):

    logid = mongolog( locals() )

    if not os.path.isfile(filepath):
```

```
        return command_error( returncode=15, stderr="Specified file is
            not a file: "+filepath , logid=logid )

    try:
        if os.path.isdir(dst):
            if dst[-1] is '/':
                dst = dst + os.path.basename(src)
            else:
                dst = dst + '/' + os.path.basename(src)

        copyfile(src, dst)
    except FileNotFoundError:
        return command_error( returncode=10, stderr='File to rename not
            found: "'+path+'"', logid=logid )

    return command_success( logid=logid )
```

Nasce per copiare un file sul sistema. ATTENZIONE: Usare con cautela questa funzione in quanto potrebbe sovrascrivere file già esistenti.

Parametri Accetta due parametri:

- `src`: è il file che deve essere copiato. Ovviamente deve essere inclusa nella stringa di questa variabile il percorso del file stesso
- `dst`: è il file di destinazione su cui il file originale verrà copiato. La variabile può anche non includere il nome dell nuovo file e in questo caso questo sarà lo stesso del file originale. Questo comportamento è lo stesso del comando di terminale `cp`

Funzionamento Memorizza dapprima un mongolog per tenere traccia dell'operazione, e controlla se il file passato come argomento è effettivamente un file regolare. Entrando nel `try` se `dst` è una directory vuol dire che l'utente vuole mantenere il nome originale del file, quindi in base al caso viene costruito l'intero percorso del nuovo file. Viene infine effettuata la copia lanciando la funzione `copyfile` della libreria del `shutil`.

Return Restituisce il dizionario di successo col campo `logid` contenente l'object id del documento mongo contenente i nomi del file originale e den nuovo file.Vedersi restituire questo dizionario vuole dire che l'operazione è andata a buon fine e che il file è stato copiato con successo.

3.1.13 readdir

```
def readdir( dirpath ):

    if not os.path.isdir( dirpath ):
        return command_error( returncode=11, stderr='Dir not found: "' +
            dirpath + '"' )

    dircontent = os.listdir( dirpath )

    return command_success( data=dircontent )
```

Serve a listare il contenuto di una directory, quindi tutti i file e le cartelle contenute.

Parametri Accetta un unico parametro che deve contenere è il percorso assoluto della cartella di cui leggerne il contenuto.

Funzionamento Dopo aver effettuato un controllo per verificare se la cartella su cui lavorare esiste lancia la funzione `os.listdir()` per ottenere la lista di file nella cartella, e restituisce questa lista al chiamante.

Return Restituisce il dizionario di successo contenente al campo data una lista di stringhe dove ogni stringa è il nome di uno dei file presente nella cartella `dirpath`.

3.2 Utenti

3.2.1 getuser

```
def getuser(user):

    try:
        command = ['getent', 'passwd',  user]
        userinfo = check_output(command, stderr=PIPE, universal_newlines=
           True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )


    #Info sull'utente dal file /etc/passwd
    userinfo = userinfo[0].split(':')


    #Getting user groups
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups #Returns the entire error dictionary as created
            by "command_error" function


    return command_success( data=dict({
        'uname': userinfo[0],
        'canlogin': 'yes' if userinfo[1]=='x' else 'no',
        'uid': userinfo[2],
        'gid': userinfo[3],
        'geco': userinfo[4].split(','),
        'home': userinfo[5],
        'shell': userinfo[6],
        'group': usergroups.pop(0), #Main user group
        'groups': usergroups if usergroups else "<␣No␣groups␣>"
```

```
    }) )
```

Questa funzione restituisce tutti le informazioni di un utente così come lette da comando `getent` e quindi dal file `/etc/passwd`.

Parametri Accetta l'unico `user` che è il nome utente dell'utente di cuisi vogliono le informazioni. Questo utente deve esistere nel sistema e deve quindi essere presente nel file `/etc/passwd`. È possibile ottenere la lista di utenti che possono essere usati per questa funzione leggendo i valori del dizionario restituito dalla funzione getusers.

Funzionamento

```
    try:
        command = ['getent', 'passwd',  user]
        userinfo = check_output(command, stderr=PIPE, universal_newlines=
            True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )

        userinfo = userinfo[0].split(':')
```

Innanzitutto viene costruito e lanciato il comando `getent passwd <user>` che ricava le informazioni sull'utente dal file `/etc/passwd`. L'esecuzione viene controllata per catturare un eventuale eccezione `CalledProcessError`. Da notare che l'output del comando viene diviso per righe dalla funzione `splitlines()` posta alla fine di `check_output()`.
Dato che le informazioni sull'utente sono divise da un due punti ma racchiusi in una stringa queste vengono separate dalla funzione `split()` che crea una lista di stringhe.

```
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups #Returns the entire error dictionary as created
            by "command_error" function
```

Visto che `getent` non restituisce la lista dei gruppo di cui l'utente fa parte ma solo il principale, ricaviamo questa lista dalla funzione getusergroups, facendo gli opportuni controlli sul codice di ritorno come spiegato in sezione 2.

```
return command_success( data=dict({
        'uname': userinfo[0],
        'canlogin': 'yes' if userinfo[1]=='x' else 'no',
        'uid': userinfo[2],
        'gid': userinfo[3],
        'geco': userinfo[4].split(','),
        'home': userinfo[5],
        'shell': userinfo[6],
        'group': usergroups.pop(0), #Main user group
        'groups': usergroups if usergroups else "<␣No␣groups␣>"
}) )
```

Non resta quindi che creare un dizinario da dare in pasto a `command_success` per essere restituita agli utenti. A parte i campi che vengono inseriti normalmente ne distinguiamo quattro che si comportano in modo diverso:

- `canlogin`: indica se è possibile effettuare l'accesso alla shell con l'utente. In particolare se il campo di `/etc/passwd` è `x` allora l'utente può effettuare l'accesso
- `geco`: Indica l'anagrafica dell'utente e altre informazioni come l'email. Questo campo è una lista ma presentatdosi come una semplice stringa divisa da virgole necessita di essere splittata prima dell'inserimento in modo da poter essere riferita direttamente
- `group`: È il gruppo principale dell'utente e viene ricavata dalla lista di gruppi in quanto primo membro, e poi rimosso da questa lista
- `groups`: È la lista dei gruppi secondari di cui l'utente fa parte e viene inserita così com'è (una lista) se il l'utente ha almeno un gruppo secondario, altrimenti viene inserita una stringa che indica l'assenza dei gruppi in modo da non vedere apparire in questo campo una lista vuota

Return Restituisce il dizionario di successo, dove al campo `data` sono presenti le informazioni sull'utente, qui descritte:

- uname: Nome utente
- canlogin: Indica la possibilità di accesso alla shell con questo utente
- uid: L'user ID dell'utente[4]
- gid: il group ID del gruppo principale di cui l'utente fa parte, che di base ha lo stesso nome dell'utente
- geco: Alcune informazioni sull'utente, ossia nome, cognome, email, stanza ecc.
- home: la home dell'utente; solitamente se l'utente non è di sistema si trova al percorso `/home/<uname>`
- shell: la shell assegnata all'utente. Di base è `/bin/bash` ma solitamente se l'utente non è di sistema si possono trovare le shell fittizie `/bin/false` e `/usr/bin/nologin`
- group: il nome del gruppo principale di cui l'utente fa parte. Solitamente alla creazione dell'utente viene creato dal sistema anche questo gruppo e gli viene dato lo stesso nome. Ad es. l'utente *giuseppe* ha come gruppo principale *giuseppe* e di base è l'unico membro
- groups: la lista dei gruppi secondari di cui l'utente faf parte. È possibile in nimodo aggiungere un utente ad uno o più gruppi usando la funzione addusertogroups

3.2.2 getusers

```
def getusers():

    with open('/etc/passwd', 'r') as opened:
        passwd = opened.read().splitlines()
```

[4]Nei sistemi UNIX based gli utenti non di sistema hanno un uid che parte da 1000 a salire.

```
    users = dict()
    for line in passwd:
        line = line.split(':', 3)
        uname = line[0]
        uid = line[2]
        users[uid] = uname

    return command_success( data=users )
```

Questa funzione nasce per ottenere la lista utenti presente nel sistema.

Parametri La funzione non prende parametri

Funzionamento

```
    with open('/etc/passwd', 'r') as opened:
        passwd = opened.read().splitlines()
```

Siccome si cerca di limitare il più possobile l'utilizzo delle funzioni della libreria `subprocess` gli utenti vengono letti dal file `/etc/passwd` con questa open; le righe di questo file vengono quindi divise ed inserite nella variabile `passwd` che diventa quindi una lista di stringhe.

```
    users = dict()
    for line in passwd:
        line = line.split(':', 3)
        uname = line[0]
        uid = line[2]
        users[uid] = uname

    return command_success( data=users )
```

Eseguito questo passaggio si itera sulle linee del dizionario `passwd`; ogni riga viene splittata di tre elementi in quanto `uname` si trova nel primo campo mentre `uid` si trova nel terzo, e vengono inserite in dizionario in cui la chiave è l'uid mentre il valore è lo uname. Questo è il dizionario da essere restituito all'utente, che viene quindi passato alla funzione di uscita `command_success`.

Return Restituisce il dizionario di successo con al campo `data` un dizionario dove la chiave è l'uid dell'utente, mentre il valore è la sua username.

3.2.3 getgroups

```
def getgroups(namesonly=False):

    with open('/etc/group', 'r') as opened:
        etcgroup = opened.read().splitlines()


    groups = list()
    if namesonly:
        groups = list(map( lambda line: line.split(':')[0], etcgroup ))
    else:
```

```
        for line in etcgroup:
            line = line.split(':')
            groups.append({
                'gname': line[0],
                'gid': line[2],
                'members': line[3].split(',')
            })

    return command_success( data=groups )
```

Questa funzione agisce nello stesso modo di `getusers()` restituendo però i gruppi del sistema invece che gli utenti.

Parametri La funzione accetta un unico parametro `namesonly` che di base è `False` e che se impostato a `True` restutuisce il solo nome dei gruppi.

Funzionamento Agisce come getusers e cioè inizialmente legge la lista dei gruppi dal file `/etc/group`, lo divide nelle sue righe e lo memorizza nella variabile `etcgroup` che diventa quindi una lista di stringhe.

```
    groups = list()
    if namesonly:
        groups = list(map( lambda line: line.split(':')[0], etcgroup ))
```

Eseguito questo passaggio dichiara la lista `groups` che andrà ritornata e verifica il valore della variabile `namesonly`. Se questo è `True` applica una funzione inline `lambda` e per ogni elemento della lista creata applica uno split dei suoi campi e ne memorizza il primo valore nella lista da restituire `groups`. In questo modo quindi `groups` sarà una lista di stringhe in cui ogni stringa è il nome di un gruppo.

```
else:
    for line in etcgroup:
        line = line.split(':')
        groups.append({
            'gname': line[0],
            'gid': line[2],
            'members': line[3].split(',')
        })

return command_success( data=groups )
```

Se invece `namesonly` è `False` si entra in un for in cui per ogni riga del dizionario `etcgroup`:

1. la riga stessa viene splittata nei suoi campi che vengono memorizzati nella variabile `line`
2. Viene creato un dizionario che contiene nome, identificativo, e membri appartenenti al gruppo
3. La lista di dizionari creata (`groups`) viene passata alla funzione di ritorno `command_success`

Return Restituisce il dizionario di successo con al campo `data` una lista di dizionari contenente le informazioni sui singoli gruppi presenti nel sistema. Ogni dizionario è costituito in questo modo:

- gname: il nome del gruppo

- gid: l'identificativo del gruppo
- members: una lista di stringhe che contiene tutti i membri del gruppo. Notare che nel codice questa lista è stata creata splittando il componente numero 3 della riga per la virgola

3.2.4 getusergroups

```
def getusergroups(user):

    command = ['groups', user]

    try:
        usergroups = check_output(command, stderr=PIPE,
            universal_newlines=True).splitlines()
    except CalledProcessError as e:
        command_error( e, command )

#                   .------Removing username from list
#                   |                          .----------.
#                   V                          V          |
    usergroups = re.sub('^.*: ', '', usergroups[0]) #Only first line
        contains the groups
    usergroups = usergroups.split(' ')

    return command_success( data=usergroups )
```

Restituisce la lista di gruppi di cui l'utente fa parte.

Parametri L'unico parametro che prende è `user` che è il nome utente di cui si vogliono conoscere i gruppi.

Funzionamento

```
    command = ['groups', user]

    try:
        usergroups = check_output(command, stderr=PIPE,
            universal_newlines=True).splitlines()
    except CalledProcessError as e:
        command_error( e, command )
```

In questa prima parte viene composto e lanciato il comando `groups` che restituisce la lista di gruppi di cui l'utente fa parte. Con la funzione `splitlines()` viene poi diviso l'output in righe in quanto nell'infausto caso il comando genera più righe a noi server solamente la prima.
Viene generata così una lista di dizionari che viene memorizzata in `usergroups`.
Siccome a noi interessa l'output del comando viene lanciata la funzione `check_output` invece di `check_call`.
Anche in questo caso viene controllato che non venga generata una eccezione `CalledProcessError`.

```
    usergroups = re.sub('^.*: ', '', usergroups[0]) #Only first line
        contains the groups
```

```
    usergroups = usergroups.split('␣')

    return command_success( data=usergroups )
```

L'output di `groups` include prima della lista lo username dell'utente, quindi prima di proseguire deve essere rimosso. Questo viene fatto con la libreria sub che con un espressione regolare elimina tutto quello che c'è prima e lo spazio che c'è dopo il due punti.
Fatta questa operazione basta quindi splittare la stringa restituita che divide i gruppi con uno spazio e chiamare la funzione `command_success` passadogli la lista di gruppi

Return Il dizionario di successo contenente al campo `data` una lista di stringhe in cui ogni stringa è un gruppo di cui l'utente `user` fa parte.

3.2.5 getusernotgroups

```
def getusernotgroups(user):

    #Getting all system groups
    groups = getgroups(namesonly=True)
    if groups['returncode'] is 0:
        groups = groups['data']
    else:
        return groups

    #Getting user specific groups
    usergroups = getusergroups(user)
    if usergroups['returncode'] is 0:
        usergroups = usergroups['data']
    else:
        return usergroups

    usernotgroup = list(filter( lambda group: not any(s in group for s in
        usergroups), groups ))

    return command_success( data=usernotgroup )
```

Nasce per completare la funzione getusergroups e al contrario di questa restituisce tutti i gruppi di cui l'utente non fa parte.
A cosa può mai servire questa funzione? Durante l'aggiunta di un utente ad un gruppo si deve avere la necessità di sapere l'utente di quali gruppi fa parte e di quali non fa part per rendere l'interfaccia può chiara ed agevole, più facile da usare e anche per garantire il corretto funzionamento dell'applicativo, riducendo le situazioni di eccezione in caso di situazioni che non avevamo programmato.

Parametri Prende l'unico parametro `user` che è il nome utente da usare per ricavare la lista dei gruppi di cui l'utente stesso non fa parte.

Funzionamento

```
#Getting all system groups
groups = getgroups(namesonly=True)
if groups['returncode'] is 0:
    groups = groups['data']
else:
    return groups

#Getting user specific groups
usergroups = getusergroups(user)
if usergroups['returncode'] is 0:
    usergroups = usergroups['data']
else:
    return usergroups
```

Per ricavare la lista di gruppi di cui l'utente non fa parte si è esegue la sottrazione tra tutti i gruppi del sistema e tutti i gruppi di cui l'utente fa parte.
In questa prima parte quindi chiamando le funzioni `getgroups()` e `getusergroups()` si ricavano rispettivamente tutti i gruppi di sistema e tutti i gruppi di cui l'utente fa parte. Sul dizionario restituito si effettuano delle verifiche per vedere se l'operazione è andata a buon fine.

```
usernotgroup = list(filter( lambda group: not any(s in group for s in
    usergroups), groups ))

return command_success( data=usernotgroup )
```

Per la sottrazione utilizziamo la funzione `filter`. Questa funzione valuta l'espressione che gli si da, se questa restituisce `True` allora inserisce l'oggetto nella lista che sta costruendo, altrimenti lo omette. Ad es. in questo caso per ogni `group` oggetto di `groups` e per ogni gruppo `s` in `usergroups` se `s` è uguale a `group` (si è usato `in` in questo caso) viene restituito `True` che viene negato a `False` e quindi il `group` non viene inserito nella lista che la funzione sta costruendo.
Le funzioni usate sono quindi le seguenti:

- `ffilter`: filtra i risultati in base all'espressione che gli si da. Se `True` li inserisce nella lista che sta costruendo, altrimenti li omette
- `any`: restituisce `True` se almeno una delle espressioni al suo interno restituisce `True`. In questo caso se almeno uno dei gruppi dell'utente coincide col gruppo `s`. Questa espressione viene negata usando `not`, quindi si ricavano tutti i gruppi che sono nella lista dei gruppi di sistema `groups` ma non nella lista dei gruppi dell'utente

Una volta effettuata la sottrazione chiama la funzione di ritorno che indica *successo* `command_success` passandogli la lista dei gruppi ricavati.

Return Il dizionario di successo contenente al campo `data` una lista di stringhe in cui ogni stringa è un gruppo di cui l'utente `user` non fa parte.

3.2.6 addusertogroups

```
def addusertogroups(user, *groups):

    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )

    try:
        for group in groups:
            command = ['adduser', user, group]
            check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

La funzione nasce per aggiungere un utente ad uno o più gruppi di sistema. Ovviamente l'utente non deve appartenere al/ai gruppo/i a cui si sta aggiungendo. È possibile ottenere una lista di questi gruppi utilizzando la funzione getusernotgroups.

Parametri

- `user`: la prima non poteva che essere l'utente che si vuole aggiungere ai gruppi
- `*groups`: la seconda è una variabile accumulativa di Python, ciò significa che dopo aver passato `user` tutto ciò che si passa dopo viene assemblato in questa variabile per poi essere scorsi un pò alla volta. Questo ci da la possibilità di poter aggiungere l'utente a quanti gruppi si vuole chiamando questa funzione una sola volta

Funzionamento

```
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
```

Essendo questa una opoerazione sensibile che va a modificare una parte del sistema si va prima di tutto a memorizzare un mongolog per tenere traccia dell'operazione. In questo caso oltre a memorizzare i parametri con cui è stata chiamata (ricavati usando la funzione `locals()`) si ricavano le informazioni sull'utente chiamando la funzione getuser e si inseriscono nel log.

```
    try:
        for group in groups:
            command = ['adduser', user, group],
```

```
            check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

L'aggiunta dell'utente ai gruppi avviene in un ciclo `for` dove per ogni gruppo passato a `*groups` viene lanciato il comando `adduser` tramite una `check_call`. Notare che in questo caso non ci interessa l'output del comando ma solo il suo codice di ritorno, è per questo che usiamo `check_call`. Se l'operazione va in eccezione come sempre di chiama la funzione `command_error` mentre se va a buon fine si chiama `command_success`, questa volta passandogli il solo logid. Il dizionario restituito all'utente avrà quindi la variabile `Data` a `None`, mentre la variabile `logid` conterrà il l'object id dell'operazione.

Return Restituisce il dizionario di successo con il campo `data` nullo e il campo `logid` contenente l'object id del documento mongo contenente le specifiche della operazione effettuata.

3.2.7 removeuserfromgroups

```
def removeuserfromgroups(user, *groups):

    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )

    try:
        for group in groups:
                command = ['gpasswd', '-d', user, group]
                check_call(command)
    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success( logid=logid )
```

Questa funzione funzione esattamente al contrario di addusertogroups, ossia permette di specificare uno i più gruppi da cui l'utente deve essere rimosso. È possibile ricavare una lista di questi gruppi utilizzando la funzione getusergroups.

Parametri

- `user`: l'utente che si vuole aggiungere ai gruppi
- `*groups`: è una variabile accumulativa di Python, ciò significa che dopo aver passato `user` si possono passare quanti gruppo si vuole e verranno sempre unificati in questa variabile, per poi essere scorsi un pò alla volta. Questo ci da la possibilità di poter rimuovere l'utente da quanti gruppi si vuole chiamando questa funzione una sola volta

Funzionamento

```
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
```

Essendo questa una opoerazione sensibile che va a modificare una parte del sistema si va prima di tutto a memorizzare un mongolog per tenere traccia dell'operazione. In questo caso oltre a memorizzare i parametri con cui è stata chiamata (ricavati usando la funzione `locals()`) si ricavano le informazioni sull'utente chiamando la funzione getuser e si inseriscono nel log.

```
    try:
        for group in groups:
                command = ['gpasswd', '-d', user, group]
                check_call(command)
    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success( logid=logid )
```

La rimozione dell'utente dai gruppi avviene in un ciclo `for` dove per ogni gruppo passato a `*groups` viene lanciato il comando `gpasswd` tramite una `check_call`. Notare che in questo caso non ci interessa l'output del comando ma solo il suo codice di ritorno, è per questo che usiamo `check_call`. Se l'operazione va in eccezione come sempre di chiama la funzione `command_error` mentre se va a buon fine si chiama `command_success`, questa volta passandogli il solo logid. Il dizionario restituito all'utente avrà quindi la variabile `Data` a `None`, mentre la variabile `logid` conterrà il l'object id dell'operazione.

Return Restituisce il dizionario di successo con il campo `data` nullo e il campo `logid` contenente l'object id del documento mongo contenente le specifiche della operazione effettuata.

3.2.8 updatepass

```
def updateuserpass(user, password):

    localsvar = locals()
    del localsvar['password']
    logid = mongolog( localsvar )

    try:
        command = ['echo', user + ':' + password]
        p1 = Popen(command, stdout=PIPE)
        command = ['/usr/sbin/chpasswd']
        p2 = Popen(command, stdin=p1.stdout)
        p1.stdout.close()

    except CalledProcessError as e:
```

```
        return command_error(e, command, logid)


    return command_success( logid=logid )
```

La funzione nasce per aggiornare la password dell'utente indicato.

Parametri

- `user`: Il nome utente dell'utente a cui cambiare la password
- `password`: La nuova password dell'utente. Notare che non serve la vecchia password in quanto l'applilcativo opera con privilegi di root

Funzionamento

```
    localsvar = locals()
    del localsvar['password']
    logid = mongolog( localsvar )
```

Essendo questa una operazione sensibile si deve prima di tutto creare un mongolog per tenere traccia dell'operazione. In questo caso però non possiamo memorizzare il parametro `password` in mongolog ne in chiaro e ne criptato in quanto sui sistemi unix-based anche se l'utenza root ha i privilegi per modificare le password di tutti gli utenti non può e non deve conoscere le password di questi. Quindi come prima cosa eliminiamo si crea una lista con i parametri chiamando la funzione `locals()` e poi si crea il mongolog.

```
    try:
        command = ['echo', user + ':' + password]
        p1 = Popen(command, stdout=PIPE)
        command = ['/usr/sbin/chpasswd']
        p2 = Popen(command, stdin=p1.stdout)
        p1.stdout.close()

    except CalledProcessError as e:
        return command_error(e, command, logid)


    return command_success( logid=logid )
```

Dopo la memorizzazione del momgolog si passa all'esecuzione dei comando necessari. Il comando per cambiare la password di un utente in modo non interattivo prevede l'utilizzo di una pipe. Con la prima esecuzione quindi si stampa sullo STDOUT la stringa `<username>:<password>'`, tale stringa viene poi catturata dal comando `/usr/sbin/chpasswd` tramite la direttiva `stdin=p1.stdout` che la parsa e cambia la password dell'utente.
Se si va in eccezione (ad esempio si passa un utente non esistente) viene creato il dizionario di command_error e resttuito all'utente, mentre se l'operazione termina correttamemnte viene creato il dizionario di command_success e restituito all'utente.

Return Viene restituito il dizionario di successo generayo da `command_success` con il campo `Data` nullo e il campo `logid` contente l'object id del mongolog.

3.2.9 getshells

```
def getshells():

    with open('/etc/shells') as opened:
        shells = opened.read().splitlines()

    #Removing comment lines
    shells = list( filter( lambda shell: not shell.startswith('#'),
       shells) )

    #Manually Appending dummy shells
    shells = shells + ['/usr/sbin/nologin', '/bin/false']

    return command_success( data=shells )
```

Alcuni delle funzioni del modulo di nomodo `user` richiedono che gli sia passata il percorso di una shell come parametro. Per portare al minimo gli errori si è creata questa funzione che non fa altro che restituire la lista delle shell installate nel sistema, in modo che l'utente non debba immettere manualmente la shell ma deve selezionarla da una lista. Un esempio è quando si chiama la funzione `updateusershell`. Evitiamo così sia errori volontari che errori di scrittura.

Parametri La funzione nella sua semplicità non prende parametri.

Funzionamento

- Apre il file che contiene le shell `/etc/shells` in lettura, ne legge il contenuto, lo divide per linee creando una lista e inserisce tale lista nella variabile `shells`
- In una funzione `filter` legge le righe una alla volta e se una riga inizia per # (cancelletto) la rimuove dalla lista. La verifica è effettuata tramite la funzione `startswith` che restituisce `True` se una stringa inizia col carattere passato come argomento
- Nella terza parte del codice allla lista delle shells aggiunge le shell dummy utilizzate per impedire l'accesso con l'utente che ha quella shell assegnata. Queste shell vengono usate per utenti di servizio, come l'utente `www-data`
- Restituisce le shell nella solita funzione di successo `command_success`

Return Restituisce il dizionario di successo con il campo `data` contenente una lista dove ogni elemento è il percorso di una delle shell installate nel sistema.

3.2.10 updateusershell

```
def updateusershell(user, shell):

    logid = mongolog( locals() )

    if not shell:
```

```
        return command_error( returncode=200, stderr="String containing
            new shell name cannot be empty", logid=logid )


    command = ['chsh', user, '-s', shell]

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

La funzione nasce per aggiornare la shell assegnata ad un utente.

Parametri

- `user`: è lo username dell'utente a cui verrà cambiata la shell
- `shell`: è la nuova shell da assegnare all'utente. La lista delle shell che si possono utilizzare può essere ricavata dalla funzione getshells

Funzionamento

```
    logid = mongolog( locals() )

    if not shell:
        return command_error( returncode=200, stderr="La stringa
            contenente il nome della shell non puo' essere vuota")
```

L'operazione risulta *sensibile*, viene quindi prima di tutto memorizzato un mongolog con le informazioni sull'operazione.
Viene poi controllato che il parametro `shell` non sia una stringa vuota, per evitare errori con la bash.

```
    command = ['chsh', user, '-s', shell]

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

Viene poi lanciato il comando `chsh` specificando nome utente e shell usando `check_call` e verificando che non sia generata una eccezione. Dopo il cambio delle shell se l'operazione è andata a buon fine quindi si chiama la funzione di successo `command_success` passandogli l'object id del mongolog.

Return Restituisce il dizionario di successo con la variabile `Data` a `None` e `logid` contenente l'object id del log su mongo contenente le informazioni sull'operazione appena effettuata.

3.2.11 adduser

```
def adduser(user, password, shell="/bin/bash"):

    logid = mongolog( locals() )

    if not shell:
        return command_error( returncode=200, stderr="String containing
            new shell name cannot be empty", logid=logid )

    try:
        command = ['useradd', '-m', '-p', password, '-s', shell, user]
        check_output(command, stderr=PIPE, universal_newlines=True)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

Serve ad aggiungere nuovi utenti al sistema.

Parametri

- `user`: è il nome utente del nuovo utente che si sta creando. Deve essere univoco nel sistema
- `password`: la password di accesso del nuovo utente
- `shell="/bin/bash"`: è la shell che verrà assegnata all'utente all'atto di creazione. Se questo parametro non viene passato viene assegnata la shell di default di sistema che è `bash`

Funzionamento

```
    logid = mongolog( locals() )

    if not shell:
        return command_error( returncode=200, stderr="La stringa
            contenente il nome della shell non puo' essere vuota" )
```

Viene creato un mongolog in quanto operazione sensibile che va a modificare il sistema e potrebbe compromettere la sicurezza dello stesso. Come in 2.2.10 viene verificato che il parametro `shell` non sia una stringa vuota per evitare errori con la bash.

```
    try:
        command = ['useradd', '-m', '-p', password, '-s', shell, user]
        check_output(command, stderr=PIPE, universal_newlines=True)
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

Viene poi composto il comando `useradd` definendo tutte le specifiche dell'utente ed indicando di creare una home col parametro `-m`, viene eseguito e viene controllato che non venga lanciata nessuna

eccezione.
Se tutto va a buon fine viene invocata la funzione di successo `command_success` passandogli l'object id del documento mongo creato contenente le informazioni sull'operazione di creazione dell'utente.

Return Restituisce il dizionario di successo generato da `command_success` con la chiave `Data` a `None` e la chiave `logid` contenente l'object id del documento creato e contenente le informazioni sull'operazione eseguita.

3.2.12 removeuser

```
def removeuser(user, removehome=False):

    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )


    try:
        command = ['deluser', user]
        if removehome: command.append('--remove-home')

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

È la funzione opposta a adduser e serve a rimuovere un utente dal sistema.

Parametri

- `user`: è il nome utente dell'utente da eliminare dal sistema
- `removehome=False`: se `True` indica di rimuovere anche la home dell'utente oltre all'utente stesso. Di default è `False` in quanto la home degli utenti potrebbero contenere dati importanti che non si vuole perdere

Funzionamento

```
    userinfo = getuser(user)
    if userinfo['returncode'] is 0:
        userinfo = userinfo['data']
    else:
        return userinfo

    logid = mongolog( locals(), userinfo )
```

La rimozione di un utente è una operazione molto importante, viene quindi generato un mongolog contenente, oltre ai parametri con cui è chiamata la funzione, anche le informazioni sull'utente che sta per essere rimosso, ricavate invocando la funzione `getuser()`.

```
    try:
        command = ['deluser', user]
        if removehome: command.append('--remove-home')

        check_output( command, stderr=PIPE, universal_newlines=True )
    except CalledProcessError as e:
        return command_error( e, command, logid )


    return command_success( logid=logid )
```

All'atto del lancio del comando `deluser` viene quindi verificato il parametro `removehome`; se questo risulta essere `True` viene aggiunto al comando `deluser <user>` anche il parametro `--remove-home` istruendo quindi lo stesso a rimuovere anche la cartella `/home/<user>/`.
Viene controllato quindi che non venga generata l'eccezione `CalledProcessError` e se tutto va a buon fine viene invocata la funzione di successo `command_success` passandogli l'object id del documento mongo che tiene traccia delle informazioni dell'operazione.

Return Restituisce il dizionario di successo generato da `command_success` con la key `Data` a `None` e la key `logid` contenente l'object id del documento mongo creato e che tiene traccia delle informazioni sull'operazione.

3.3 Applicazioni

La libreria nasce per la gestione dei pacchetti sul sistema. nomodo nasce su un sistema Ubuntu server 16.04 cioè Debian-based ed attualmente supporta solamente la gestione dei pacchetti tramite i tool di questa categoria di sistemi, ossia `dpkg` e `apt` (evoluzione di `apt-get`). Da notare però che nelle funzione non viene mai utilizzato il comando `apt` ma sempre il vecchio `apt-get` in quanto come suggerito dagli sviluppatori non ancora pronto per l'uso negli script.

3.3.1 aptupdate

```
def aptupdate():

    logid = mongolog( locals() )

    try:
        command = ['apt-get', 'update']
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

Diversamente dai sistemi rpm-based nei sistemi debian-based la cache dei pacchetti va aggiornata a mano. La cache dei pacchetti è fondamentale al sistema per eseguire ricerche e installazioni di nuovi pacchetti più rapidamente possibile. Questa funzione nasce proprio per aggiornare tale cache.

Parametri La funzione non prende parametri.

Funzionamento Viene semplicemente costruito e lanciato il comando `apt-get update` che come detto aggiorna la cache dei pacchetti, dopo aver creato un mongolog.

Return Restituisce il dizionario di successo contenente al campo `logid` l'object id del documento mongo (o mongolog in nomodo) contenente le specifiche dell'operazione.

3.3.2 listinstalled

```
def listinstalled( summary=False ):

    options = '-f=${binary:Package};${Version};${Architecture}' + ( ';${
       binary:Summary}\n' if summary else '\n' )
    command = ['dpkg-query', options, '-W']

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
           True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )
    except FileNotFoundError as e:
        return command_error( e, command )


    #Lista di chiavi per le informazioni sull'app
    keys = ['name', 'version', 'architecture']
    if summary: keys.append('summary')

    #Conterra' una lista di dizionari con tutte le app installate nel
       sistema
    pkgs = list()

    #Inserimento valori nella lista apps()
    for i in output:
        appinfo = i.split(';')
        pkgs.append( dict( zip(keys, appinfo) ) )

    return command_success( data=pkgs )
```

La funzione nasce per restituire la lista dei pacchetti installata nel sistema.

Parametri Accetta l'unico parametro `summary` di default a `False` che se impostato a `True` indica alla funzione di restituire nell'output anche la descrizione breve sulle funzionalità della singola applicazione.

Funzionamento Utilizzando il comando `dpkg-query` le opzioni di visualizzazione sono leggermente complicate, è possibile visualizzarle nella variabile `options`; queste informazioni servono ad ottenere architettura, versione, nome dell'applicazione e probabilmente la descrizione della stessa se il parametro

`summary` è `True`.
Viene quindi costruito il comando e lanciato tramite la `check_output`, verificando che non venga lanciata l'eccezione `CalledProcessError`. Una volta lanciato si ottiene quindi una lista di informazioni sulle applicazioni installate ma in un formato in cui i campi sono divisi da un punto e virgola e senza indicazioni su cosa rappresenta lo stesso. Si desidera un output in le informazioni della singola applicazione sono contenute in un dizionario; si crea quindi una lista di chiavi che saranno quelle del dizionario, si dividono i campi delle app e si concatenano alle chiavi usando la funzione `zip()`; il dizionario di ritorno è quello desiderato e viene quindi restituito al frontend o all'utente chiamante.

Return Restituisce il dizionario di successo generato dalla funzione `command_success()` in cui il campo `data` contiene una lista di dizionari in cui ogni dizionario contiene le stesse informazioni per tutte le app installate. Le informazioni riguardano *architecture* cioè l'architettura per cui l'app è stata creata, *name* che è il nome dell'applicazione e *version* che è la versione dell'applicativo. Un esempio della lista di dizionario è il seguente:

```
.............

{'architecture': 'all',
  'name': 'python3-pkg-resources',
  'version': '20.7.0-1'},
{'architecture': 'all', 'name': 'python3-prettytable', 'version': '
   0.7.2-3'},
{'architecture': 'all',
  'name': 'python3-problem-report',
  'version': '2.20.1-0ubuntu2.18'},
{'architecture': 'all', 'name': 'python3-pyasn1', 'version': '0.1.9-1'},
{'architecture': 'amd64',
  'name': 'python3-pycurl',
  'version': '7.43.0-1ubuntu1'},

.............
```

3.3.3 aptsearch

```
def aptsearch( pkgname, namesonly=True ):

    #Cannot search on empty string
    if not pkgname:
        command_error( returncode=255, stderr='Empty search string not
            allowed' )

    command = ['apt-cache', 'search', pkgname]
    if namesonly: command.append('--names-only')

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True).splitlines()
```

```
    except CalledProcessError as e:
        return command_error( e, command )

    keys = ['name', 'desc']
    pkgs = list()

    for i in output:
        appinfo = i.split(' - ')
        pkgs.append( dict( zip(keys, appinfo) ) )

    return command_success( data=pkgs )
```

È ideata per effettuare la ricerca dei pacchetti disponibili nelle repository installate nel sistema. Verrebbe quindi probabilmente usata prima della funzione aptinstall per cercare il pacchetto che si desidera installare.

Parametri Accetta due parametri:

- `pkgname`: è la stringa con cui effettuare la ricerca
- `namesonly=True`: è la modalità di ricerca: di default viene cercato solo nel nome del file, se invece si setta a `False` viene cercato anche in altri campi dei pacchetti quali la decsrizione

Funzionamento La prima operazione è verificare che l'utente non abbia inserito una stringa vuota e ritornare un codice di ritorno 255 in tale caso. Tale controllo viene fatto anche dal frontend ma la sicurezza non è mai troppa.
Viene quindi costruito il comando appenendogli il parametro `--names-only` se l'omonimo parametro è `True` e lanciato immagazzinando lo standard output nella variabile `output`, che diventa quindi una lista di stringhe (perchè usiamo la splitlines()).
L'output restituito è nel formato "nome_pacchetto - descrizione" ma siccome noi vogliamo un dizionario come prima cosa creiamo le chiavi immagazzinandole nella variabile `keys` dividiamo quindi ogni riga dell'output usando `split()` ricavando nome e descrizione e concateniamo infine chiavi e valori usando la funzione `zip()`. Siccome per ogni app è stato creato un dizionario avremo quindi che la variabile `appinfo` sarà una lista di dizionari; questa variabile viene quindi restituita.

Return Restistuisce il dizionario di successo, contenente al campo `data` una lista di dizionari dove ogni dizionario contiene nome e descrizione di una delle applicazioni che corrispondono alla ricerca.

3.3.4 aptshow

```
def aptshow(pkgname, onlydependences=False):

    mode = 'depends' if onlydependences else 'show'

    try:
        command = ['apt-cache', mode, pkgname]
        output = check_output(command, stderr=PIPE, universal_newlines=
            True)
    except CalledProcessError as e:
```

```
        return command_error( e, command )


    if onlydependences:
        #Remove the first line (header)
        toreturn = re.sub('^.*\\n', '', output)
    else:
        #On multiple results only keep the first one
        output = output.split('\n\n')[0]
        output = output.splitlines() #<-- We use splitlines() here
            because onlydependences does not need a split-lined output

        #Check whether the package is installed or not
        isinstalled = None
        try:
            command = ['dpkg', '-l', pkgname]
            check_call(command)
        except CalledProcessError as e:
            isinstalled = False
        if isinstalled is None: isinstalled = True

        #Removing useless lines
        linestomantain = ['Package:', 'Version:', 'Priority:', 'Section:'
            , 'Origin:', 'Installed-Size:', 'Depends:', 'Description', ' '
            ]
        output = list( filter( lambda line: any( line.startswith(s) for s
            in linestomantain), output ) )

        #Merging all of descrition lines
        i = 0
        n = len(output)

        while i < n:
            if output[i].startswith(' '):
                output[i-1] = output[i-1] + output[i] #<-- Merge lines
                del output[i] #<-- Delete current line
                n -= 1
            else:
                i += 1


        #Converting list to dictionary
        toreturn = dict()
        for line in output:
            dictelems = line.split(':', maxsplit=1)
            toreturn.update({ dictelems[0] : dictelems[1] })

        #Is this package installed?
        toreturn.update({ 'Intalled' : 1 if isinstalled else 0 })
```

```
    return command_success( data=toreturn )
```

La funzione nasce per mostrare le specifiche di un pacchetto, quali versione, autore, descrizione ecc.

Parametri Accetta due parametri:

- `pkgname`: è il nome del pacchetto di cui ricavare e restituire le informazioni
- `onlydependences=False`: se questa variabile viene settata a `True` vengono rstituite solo le informazioni sulle dipendenze del pacchetto stesso

Funzionamento

```
    mode = 'depends' if onlydependences else 'show'

    try:
        command = ['apt-cache', mode, pkgname]
        output = check_output(command, stderr=PIPE, universal_newlines=
            True)
    except CalledProcessError as e:
        return command_error( e, command )
```

La prima parte è una parte comune della funzione, quindi viene eseguita sia se si stanno chiedendo solo le dipendenze del pacchetto sia se si voglino tutte le informazioni. Viene decisa l'operazione da eseguire a seconda del valore del parametro `onlydependences`. Si esegue quindi il comando che memorizzerà nella variabile `output` le informazioni desiderate in un formato rozzo. cioè un'unica stringa contenente le informazioni.

```
    if onlydependences:
        #Remove the first line (header)
        toreturn = re.sub('^.*\\n', '', output)
```

Si entra poi nell'if che esegue un codice od un altro. Quindi se `onlydependences` è `True` viene semplicemente eliminata la prima riga che contiene l'header e ritornata la variabile `output`.

```
    else:
        #On multiple results only keep the first one
        output = output.split('\n\n')[0]
        output = output.splitlines() #<-- We use splitlines() here
            because onlydependences does not need a split-lined output

        #Check whether the package is installed or not
        isinstalled = None
        try:
            command = ['dpkg', '-l', pkgname]
            check_call(command)
        except CalledProcessError as e:
            isinstalled = False
        if isinstalled is None: isinstalled = True
```

Se invece si vogliono le informazioni sul pacchetto si inizia da questo pezzo di codice qui sopra. Siccome ci interessano le informazioni sul pacchetto in gerenale e non su una versione specifica vengono divise le informazioni dei diversi pacchetti tramite la `split()` e vengono mantenute le informazioni solo sul primo di questo che contiene la versione più recente dello stesso.
Viene quindi diviso l'output per righe (`splitlines()`) in quanto più facile da manipolare in questo modo.
Viene quindi lanciato un secondo comando che mira a verificare se il pacchetto è installato nel sistema. Viene quindi usato `dpkg` e memorizzata questa informazione nella variabile `isinstalled`.

```
        #Removing useless lines
        linestomantain = ['Package:', 'Version:', 'Priority:', 'Section:'
            , 'Origin:', 'Installed-Size:', 'Depends:', 'Description', '␣'
            ]
        output = list( filter( lambda line: any( line.startswith(s) for s
             in linestomantain), output ) )

        #Merging all of descrition lines
        i = 0
        n = len(output)

        while i < n:
            if output[i].startswith('␣'):
                output[i-1] = output[i-1] + output[i] #<-- Merge lines
                del output[i] #<-- Delete current line
                n -= 1
            else:
                i += 1
```

Molto delle informazioni ottenute non ci servono, vengono quindi rimosse le linne inutili utilizzando la funzione `filter()` che si vede in codice.
Analizzando l'output restituito ci rendiamo conto che questo è simile ad un dizionario a parte per il fatto che alcuni campi hanno il valore su più righe. Ad esempio per la chiave `Description` abbiamo il seguente valore: `Description: Vi IMproved - enhanced vi editor Vim is an almost compatible version of the UNIX editor Vi. . Many new features have been added: multi level undo, syntax highlighting, command line history, on-line help, filename completion, block operations, folding, Unicode support, etc. . This package contains a version of vim compiled with a rather standard set of features. This package does not provide a GUI version of Vim. See the other vim-* packages if you need more (or less).` Per usare questa informazione quindi dobbiamo prima di tutto unire tutte le righe in una sola. Si può notare che tutte le righe che sono il continuo di quella precedente sono precedute da uno spazio. Quindi proprio in base a questa particolarità possiamo distringuere le righe che contengono na chiave da quelle che sono il continuo della riga precedente.
È stato quindi creato l'algoritmo che si vede in codice per ogni riga dell'output se la riga corrente contiene uno spazio iniziale (`startswith(' ')`) esegue il merge di questa riga con precedente e la elimina dopo la copia.

```
        #Converting list to dictionary
        toreturn = dict()
        for line in output:
            dictelems = line.split(':', maxsplit=1)
```

```
            toreturn.update({ dictelems[0] : dictelems[1] })

        #Is this package installed?
        toreturn.update({ 'Intalled' : 1 if isinstalled else 0 })


    return command_success( data=toreturn )
```

Uno volta aggiunstato l'output è ora di convertirlo in un dizionario chiave valore, e viene fatto tramite il `for` che si vede in codice. Viene alla fine aggiunto il campo `Installed` che indica se il pacchetto è installato o meno nel sistema usando la variabile `isinstalled` ricavata prima. Viene quindi alla fine restituito il dizionario creato che si presenta in questo aspetto:

```
{'Depends': ' nginx-core (>= 1.10.3-0ubuntu0.16.04.2) | nginx-full (>= '
            '1.10.3-0ubuntu0.16.04.2) | nginx-light (>= '
            '1.10.3-0ubuntu0.16.04.2) | nginx-extras (>= '
            '1.10.3-0ubuntu0.16.04.2), nginx-core (<< '
            '1.10.3-0ubuntu0.16.04.2.1~) | nginx-full (<< '
            '1.10.3-0ubuntu0.16.04.2.1~) | nginx-light (<< '
            '1.10.3-0ubuntu0.16.04.2.1~) | nginx-extras (<< '
            '1.10.3-0ubuntu0.16.04.2.1~)',
 'Description-en': ' small, powerful, scalable web/proxy server Nginx ("
    engine '
                   'X") is a high-performance web and reverse proxy '
                      server '
                   'created by Igor Sysoev. It can be used both as a '
                   'standalone web server and as a proxy to reduce the '
                      load on '
                   'back-end HTTP or mail servers. . This is a dependency
                      '
                   'package to install either nginx-core (by default), '
                   'nginx-full, nginx-light, or nginx-extras.',
 'Description-md5': ' 2d277b9313aa50e3bfd675e64b49532c',
 'Installed-Size': ' 37',
 'Intalled': 1,
 'Origin': ' Ubuntu',
 'Package': ' nginx',
 'Priority': ' optional',
 'Section': ' web',
 'Version': ' 1.10.3-0ubuntu0.16.04.2'}
}
```

Return Restituisce il dizionario di successo (se la funzione è terminata con successo) contenente al campo `data` una dizionario contenente le informazioni sul pacchetto se `onlydependences=False` (inclusa l'informazione indicante se il pacchetto è installato o meno) o una stringa contenente le dipendenze del pacchetto stesso se il parametro `onlydependences` è `False`.

3.3.5 aptinstall

```
def aptinstall(pkgname):

    logid = mongolog( locals(), {'dependencies' : aptshow(pkgname,
       onlydependences=True)} )

    command = ['apt-get', 'install', '-y', pkgname]
    environ = {'DEBIAN_FRONTEND': 'noninteractive', 'PATH': os.environ.
       get('PATH')}

    try:
        check_call( command, env=environ ) #, stdout=open(os.devnull, '
           wb'), stderr=STDOUT)
    except CalledProcessError:
        return command_error( returncode=14, stderr='Package installation
           error. Package name: "'+pkgname+'"', logid=logid )

    return command_success( logid=logid )
```

Utile all'installazione di nuovi pacchetti nel sistema, che facciano parte dei repository installati su questo.

Parametri Accetta un solo parametro che è il nome del pacchetto da installare.

Funzionamento Si parte con la creazione di un mongolog per contenere le specifiche dell'operazione. Gli si passa in quest caso anche una lista di dipendenze che l'installazione porta con se.
Si passa quindi alla creazione del comando. Notare che in questo caso si è anche costruito un dizionario `environ` in quanto non solo l'installazione da python tramite `check_call()` necessita che gli sia specificato dove prendersi il PATH di sistema, ma necessita anche che gli sia detto che il processo sia *non interattivo* in quanto sarebbe impossibile interagire.
Si lancia quindi il comando come al solito passanfogli anche l'environment e restituendo errore in caso venga lanciata l'eccezione `CalledProcessError`.

Return Restituisce il dizionario di successo (se la funzione è stata eseguita senza errori) con il campo `data` vuoto e il campo `logid` contenente l'object id del mongolog, ossia del documento mongo contenente le specifiche dell'operazione. Vedersi restituire questo dizionario indica che il pacchetto è stato installato nel sistema con successo.

3.3.6 aptremove

```
def aptremove(pkgname, purge=False):

    logid = mongolog( locals(), {'dependencies' : aptshow(pkgname,
       onlydependences=True)} )

    command = ['apt-get', 'purge' if purge else 'remove', '-y', pkgname]
    environ = {'DEBIAN_FRONTEND': 'noninteractive', 'PATH': os.environ.
       get('PATH')}
```

```
    try:
        check_call( command, env=environ ) #stdout=open(os.devnull, 'wb')
            , stderr=STDOUT)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

Nasce per consentire la rimozione di un pacchetto dal sistema e delle sue dipendenze.

Parametri Accetta due parametri:

- `pkgname`: è il pacchetto che si vuole rimuovere dal sistema
- `purge=False`: se `True` indica che oltre al pacchetto devono essere rimosse anche tutte le sue dipendenze a patto che la rimozione di queste non intacchi con altre componenti del sistema. In tal caso la dipendenza che causerebbe problemi non viene rimossa

Funzionamento Genera prima di tutto il mongolog ugualmente a come fa la funzione aptinstall, includendo cioè in questo le dipendenze del pacchetto che si sta per rimuovere.
Viene quindi generato l'environment e il comando così come fa aptinstall (vedere questo per la spiegazione completa), lancia il comando e restituisce il dizionario di successo.

Return Restituisce il dizionario di successo (se la funzione è stata eseguita senza errori) con il campo `data` vuoto e il campo `logid` contenente l'object id del mongolog, ossia del documento mongo contenente le specifiche dell'operazione. Vedersi restituire questo dizionario indica che il pacchetto è stato rimosso dal sistema con successo, incluse le dipendenze che non causano problemi con le funzionalità del sistema, come spiegato nella sezione *Funzionamento*.

3.3.7 getexternalrepos

```
def getexternalrepos():

    repospath = '/etc/apt/sources.list.d/'
    reposfiles = os.listdir(repospath)

    #Removing file that ends with '.save'
    reposfiles = list( filter( lambda item: not item.endswith('.save'),
        reposfiles ) )

    #List to return
    repos = list()

    for filename in reposfiles:
        with open(repospath + filename) as opened:
            repos.append({
                'filename': filename,
                'lines': opened.read()
            })
```

```
    return command_success( data=repos )
```

La funzione quando lanciata restituisce la lista delle repo installate nel sistema che non siano quelle base. La gestione di queste ultime puà avvenire invece direttamente editando il file nella sezione *File* di nomodo.

Parametri Non prende nessun parametro.

Funzionamento Viene definita la cartella contenente le repo esterne (di base `/etc/apt/sources.list.d` e vengono letti i file nella stessa.
Vengono quindi eliminati dalla lista i file `.save` non utili al nostro scopo, utilizzando la funzione `filter()`. Viene quindi letto il contenuto del file ed inserito in un dizionario che viene inserito nella lista di dizionari da restituire all'utente.

Return Restituisce il dizionario di successo il cui campo `data` contiene una lista di dizionari dove ogni dizionario contiene il nome e il contenuto di uno dei file che contiene le repo esterne.

3.3.8 getreponame

```
def getreponame(): return 'nomodo-' + datetime.datetime.now().strftime('%
    Y%m%d%H%M%S')
```

Genera un nome da usare per un nuovo file di repo e funziona allo stesso modo di getcronname. Viene chiamato dal frontend per riempire il campo *name* all'aggiunta di un nuovo repository. L'utente ovviamente ha la possibiltà di cammbiare questo nome prima dell'aggiunta dello script.

Parametri Non prende nessun parametro.

Funzionamento Semplicemente contatena la stringa "nomodo-" alla data e ora odierna col formato che si vede nel codice.

Return È una delle pochissime funzione che non chiama una delle due funzioni di ritorno di nomodo. Restituisce la stringa costruita come spiegato nel funzionamento.

3.3.9 addrepo

```
def addrepo(content, name):

    logid = mongolog( locals() )

    filename = '/etc/apt/sources.list.d/' + name + '.list'
    repofile = open( filename, 'a')
    repofile.write(content + '\n')
    repofile.close()

    return command_success( logid=logid )
```

Nasce per consentire all'utente l'aggiunta di un nuovo repository usando l'interfaccia di nomodo.

Parametri Accetta due parametri:

- `content`: è la stringa (anche multiriga) che contiene l'url e le specifiche del nuovo repository. Un esempio per le repo per l'installazione dell'ultima versione di MariaDB è il seguente: `deb [arch=amd64,i386,ppc64el] http://mirror.netcologne.de/mariadb/repo/10.3/ubuntu xenial main`
- `name`: è il nome del file che conterrà l'url del nuovo repository. In caso il file sia già esistente la nuova repository verrà aggiunta al file già esistente

Funzionamento Dopo aver creato il mongolog apre il file in modalità append aggiungendogli l'url e ritorna il dizionario di successo. In modalità append se il file non esiste viene creato.

Return Viene restituito il dizionario di successo ad indicare la buona riuscita dell'operazione, con il campo `data` vuoto e con il campo `logid` contenente l'object id del mongolog creato. È difficile che fallisca e la buona riuscita indica che la repo è stata aggiunta. Dopo l'aggiunta della repo lanciare `aptupdate` per rigenerare la cache di `apt`.

3.3.10 removerepofile

```
def removerepofile(filename):

    result = filedel( externalreposdir + filename )['logid']
    filedel( externalreposdir + filename + '.save' ) #Ignores errors if
       file not exists ignoring return dictionary

    if result['returncode'] is 0:
        return command_succes( logid=logid )
    else
        return result
```

Serve a rimuovere uno dei file delle repo e il suo *.save*. Non si ritiene necessario rimuovere i singoli url dai file in quanto i file della cartella `/etc/apt/sources.list.d/` contengono solitamente una sola repo, raramente in più varianti. È tuttavia rimuovere i singoli url da questi file (e il file principale `/etc/apt/sources.list`) alla sezione editing della pagina *File* del progetto nomodo.

Parametri Accetta un solo parametro che è il file da rimuovere.

Funzionamento Semplicemente richiama due volte la funzione filedel della sezione Utilities per rimuovere il file principale e il suo .save. Notare che siccome il dizionario di ritorno della seconda `filedel()` non viene memorizzato da nessuna parte in risultato della operazione viene ignorato. Si è proceduto in questo modo in quanto questo file potrebbe non esistere.

Return Restituisce il dizionario di successo contenente il logid del mongolog memorizzato dalla prima cancellazione se l'operazione va a buon fine e il primo file è stato cancellato correttamente. Restituisce altrimenti il dizionario di erroe generato dalla funzione `filede()` se il codice di ritorno è diverso da 0 e quindi l'operazione è fallita.

3.4 Network

La sezione network raccoglie tutte le funzioni utili per la gestione della rete, quali creazione e distruzione di interfacce, cambio indirizzo, settaggio rotte ecc.

3.4.1 ifacestat

```
def ifacestat(iface="", namesonly=False):

    command = ['ifconfig', '-a']
    if iface: command.append(iface)

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True)
    except CalledProcessError as e:
        return command_error( e, command )


    output = output.split('\n\n')
    del output[-1]


    ifaces = list() if namesonly else dict()
    for iface in output:

        iface = iface.splitlines()
        firstline = iface.pop(0).split(None, maxsplit=1)


        if namesonly:
            ifaces.append( firstline[0] )
        else:
            iface.insert(0, firstline[1])

            i = 0
            for index, value in enumerate(iface):
                i += 1
                iface[index] = iface[index].strip()
                if iface[index].startswith('UP') or iface[index].
                    startswith('DOWN'): break

            ifaces.update({ firstline[0]: iface[:i] })

    return command_success( data=ifaces )
```

La funzione serve ad ottenere uno sommario sullo stato delle interfacce di rete attualmente installate su sistema.

Parametri

- `iface=""`: Se non nullo (come di defaut) questo parametro serve a farsi restituire dalla funzione lo stato della sola interfaccia indicata. Questa variabile deve contenere il nome esatto dell'interfaccia di cui si vogliono le informazioni (es. "ens1")
- `namesonly=False`: se a `True` restituisce i soli nomi dell'interfaccia invece di tutte le informazioni

Funzionamento

```
command = ['ifconfig', '-a']
if iface: command.append(iface)

try:
    output = check_output(command, stderr=PIPE, universal_newlines=
        True)
except CalledProcessError as e:
    return command_error( e, command )
```

Il comando che restituisce l'output più adatto è `ifconfig`, viene quindi lanciato usando `check_call()` e l'output conservato nella variabile `output`. Notare che ad `ifconfig` si è aggiunto il parametro `-a` in quanto vogliamo ottenere le informazioni su tutte le interfacce e non solo sulle interfacce attualemente abilitate. Se il parametro `iface` è non nullo questa stringa viene aggiunta al comando così da ottenere solo le info di questa interfaccia.[5]

```
output = output.split('\n\n')
del output[-1]

ifaces = list() if namesonly else dict()
```

`ifconfig` divide le informazioni sulle interfacce con una linea vuota e la fine dell'output con due linee vuote. Dividiamo quindi l'output splittando per `"\n\n"` ottenendo una lista di N+1 stringhe dove N è il numero delle interfacce del sistema e 1 è la stringa vuota che eliminiamo lanciando il secondo comando (`del`).
Viene poi creata la variabile `ifaces` che conterrà le informazioni da restituire all'utente. Se il parametro `namesonly` è settato a `True` viene inizializzata come lista (una lista di nomi), come dizionario altrimenti.

```
for iface in output:

    iface = iface.splitlines()
    firstline = iface.pop(0).split(None, maxsplit=1)

    if namesonly:
        ifaces.append( firstline[0] )
    else:
        iface.insert(0, firstline[1])
        i = 0
        for index, value in enumerate(iface):
            i += 1
            iface[index] = iface[index].strip()
```

[5] Anche se `iface` è settato non viene comunque rimosso il parametro `-a` in quanto non interferisce con la buona esecuzione del comando

```
                if iface[index].startswith('UP') or iface[index].
                    startswith('DOWN'):  break

        ifaces.update({ firstline[0]: iface[:i] })

    return command_success( data=ifaces )
```

Andiamo quindi ad iterare su queste interfacce che abbiamo ricavato. Per ogni intrefaccia ricavata:

- Si dividono le informazioni sulle interfacce per riga (splitlines())
- La prima riga contiene il nome dell'interfaccia più alcune informazioni divisi da uno spazio (il primo spazio della riga); si va quindi a rimovere questa riga (pop(0)), e la si divide per il primo spazio ottenendo `firstline` che è una lista dove il primo elemento è il nome dell'interfaccia mentre il secondo è la prima riga di informazioni
- Se `namesonly` è `True` si vogliono solo i nomi delle interfacce e si inserisce in `ifaces` (usata come lista) il nome dell'interfaccia contenuta in `firstline[0]` per poi passare alla prossima iterazione
- Se invece `namesonly` è `False`, `ifaces` è un dizionario. Questo dizionario verrà creato in modo che la chiave sia il nome dell'interfaccia mentre il valore siano le sue informazioni. Andiamo quindi prima di tutto ad inserire la prima riga di informazioni (contenuta in `firstline[1]` nella variabile `iface` in modo che questa variabile contenga nuovamente la lista completa delle informazioni sull'interfaccia
- L'output di `ifconfig` porta con se delle informazioni che attualmente a noi non servono. Per eliminare queste informazioni che risiedono nelle ultime righe della variabile `iface` si esesegue questo secondo ciclo for. Il ciclo viene eseguito usando `enumerate` in quanto abbiamo bisogno di modificare la variabile `iface` originale e non una sua copia, per motivi in seguito spiegati.
  Si costituisce un indice `i` che indica la riga su cui stiamo iterando e:
  - Viene incrementato l'indice che identifica la riga sulla quale si sta iterando
  - Vengono rimossi gli spazi iniziali e finali della riga. Da qui l'uso di `enumerate`
  - Se la riga inizia per *UP* oppure *DOWN* allora abbiamo raggiunto l'ultima riga che contiene informazioni a noi utili. Si lancia quindi un break uscendo dal ciclo; la varibile `i` a questo punto contiene l'esatta riga dove fermarsi e potrà essere sfruttata per tagliare le righe che non ci servono
- Usciti dal ciclo non ci resta che tagliare le righe che non ci servono ed inserire nel dizionario `ifaces` la chiave e il valore. Viene quindi lanciato `ifacea.update()` dove si passa come chiave `firstline[0]` che è il nome dell'interfaccia e come valore `iface[:i]` che solo le informazioni sulla stessa tagliate fino alla riga `i`

Alla fine del parsing, come solito per le funzioni di nomodo, viene chiamata la funzione di successo `command_success` passandogli nel parametro `data` il dizionario appena creato `ifaces`.

Return

- Se `namesonly` è `True` restituisce una lista di stringhe dove ogni stringa è il nome di una intrefaccia presente nel sistema
- Se `namesonly` è `False` restituisce un dizionario dove la chiave è il nome delle interfacce di sistema mentre il valore sono le informazioni sulla stessa, come ad esempio indirizzo IPv4 e IPv6, MAC address, stato ecc.

3.4.2 getnewifacealiasname

```
def getnewifacealiasname(iface):

    ifaces = ifacestat( namesonly=True )
    if ifaces['returncode'] is 0:
        ifaces = ifaces['data']


    aliasid = 0
    for item in ifaces:
        if item.startswith( iface + ':' ):
            item = int(re.sub('.*:', '', item))
            if aliasid is item: aliasid += 1
            else: break

    return command_success( data = iface + ':' + str(aliasid) )
```

Questa funzione non fa altro che restituire al chiamante il possibile nome di una nuova interfaccia alias data un'interfaccia fisica esistente. È intesa per essere usata come titolo della pagina in cui l'utente andrà ad inserire le specifiche della nuova interfaccia e per essere passata alla funzione createalias per la creazione dell'interfaccia virtuale.

Parametri Prende un solo parametro `iface` che è l'interfaccia a cui fare riferimento per la creazione del nuovo alias.

Funzionamento La funzione si basa su queste 3 considerazioni:

1. Il nome di una interfaccia alias è costitita dal nome dell'interfaccia più un due punti più un identificativo numerico per l'alias stesso che parta da 0
2. Data una interfaccia, ad es. `ens1`, se per questa interfaccia sono già presenti degli alias, ad es. `ens1:1` e `ens1:2`, deve essere restituito all'utente il nome alias `ens1:3`
3. Per evitare un numero sempre crescente per i nomi alias un nuovo nome dovrà riempire il buco lasciato da un interfaccia che sia stata eliminata. Ad es. se abbiamo `ens1:0` e `ens1:2` la nuova interfaccia dovrà avere il nome `ens1:1`

Si discutono i metodi utilizzati per soddisfare tali principi.

```
    ifaces = ifacestat( namesonly=True )
    if ifaces['returncode'] is 0:
        ifaces = ifaces['data']
```

Viene chiamata la funzione ifacestat con il parametro `namesonly` impostato a `True` così da ottenere i nomi di tutte le interfacce del sistema; viene inoltre fatto un controllo sul codice di ritorno della funzione.

```
    aliasid = 0
    for item in ifaces:
        if item.startswith( iface + ':' ):
            item = int(re.sub('.*:', '', item))
```

```
            if aliasid is item: aliasid += 1
            else: break

    return command_success( data = iface + ':' + str(aliasid) )
```

Si inizializza quindi una variabile `aliasid` al primo numero possibile per gli identificativi alias, cioè 0 (ad esempio `ens1:0`).
Per capire quanto a seguire bisogna tenere in mente che l'output di ifconfig ordina gli alias di un'interfaccia; ad esempio con `ens1` l'ordine sarà `ens1, ens1:0, ens1:2` e mai ad esempio `ens1, ens1:2, ens1:0`
Con tale considerazione si inizia iterando sui nomi delle interfacce. Se questa è un alias dell'interfaccia passata come parametro, cioè se il suo nome inizia (`startswith`) con il nome dell'interfaccia `iface` seguita dal due punti si entra nell'if che si vede in codice. In questo `if` si elimina il nome dell'interfaccia ed i due punti utilizzand la libreria `re` (ad esempio `ens0:1` diventa semplicemente 1), ottenendo il numero identificativo della prima interfaccia alias; si confronta quindi questo numero con `aliasid`. Se questi sono uguali l'interfaccia alias con identificativo `aliasid` è sicuramente occupata, quindi si itera e si tenta nuovamente con il prossimo alias; se invece `aliasid` non coincide col numero identificativo del primo alias allora il posto è libero e si esce dal ciclo con un break.
Dopodichè basta chiamare la funzione di successo `command_success` passandogli nel campo `data` il nome della nuova interfaccia, che non è altro che il nome della stessa seguita dal 2 punti e dal numero identificativo dell'alias contenuto nella variabile `aliasid`.

Return Restituisce il nome del nuovo alias dell'interfaccia `iface` da utilizzare come titolo della pagina in cui l'utente andrà ad inserire le specifiche della nuova interfaccia o per essere passata alla funzione createalias per la creazione dell'interfaccia virtuale.

3.4.3 ifacedown

```
def ifacedown( iface ):

    logid = mongolog( locals() )

    command = ['ifconfig', iface, 'down']

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

La funzione è intesa per disattivare un'interfaccia di rete.

Parametri Prender un solo parametro che è l'interfaccia di rete da disattivare.

Funzionamento Essendo un'operazione sensibile prima di tutto crea un mongolog e ne memorizza l'object id nella variabile `logid`.
Poi semplicemente costruisce il comando in questo modo: `ifconfig <nome_interfaccia> down` e lo lancia usando `check_call` (in quanto non è restituiuto output) e verifica che non siano avvenute

eccezioni. Chiama all fine la funzione di successo `command_success` passandoli nel campo `logid` l'object id del documento mongo creato e contenenti le specifiche dell'operazione appena eseguita.

Return Restituisce il dizionario di successo con il campo `data` a `None` e il campo `logid` contenente l'object id del documento mongo che contiene le specifiche dell'operazione eseguita.

3.4.4 ifaceup

```
def ifaceup( iface, address="", netmask="", broadcast="" ):

    logid = mongolog( locals() )

    command = ['ifconfig', iface]
    if address: command.append(address)
    if netmask: command = command + ['netmask', netmask]
    if broadcast: command = command + ['broadcast', broadcast]
    command.append('up')

    try:
        check_output = check_call(command, stderr=PIPE,
           universal_newlines=True)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

È l'opposto della funzione ifacedown e serve ad abilitare un'intrefaccia di rete attualmente disabilitata.

Parametri Prende 4 parametri:

- `iface`: il nome dell'interfaccia da tirare sù
- `address=""`: è l'indirizzo da assegnare all'interfaccia. Non è obbligatorio in quanto l'interfaccia potrebbe già avere un indirizzo assegnato
- `netmask=""`: è la maschera di rete da assegnare all'interfaccia. Non è obbligatorio in quanto l'interfaccia potrebbe già avere una maschera di rete settata
- `broadcast=""`: è l'indirizzo facente parte della stessa rete dell'interfaccia, sulla quale i pacchetti vengono inviati in broadcast a tutti gli host della rete stessa. Non è obbligatorio in quanto l'interfaccia potrebbe già avere un indirizzo broadcast settata

Funzionamento Si memorizza prima di tutto un mongolog per tenere traccia dell'operazione.
Durante la fase di costruzione del comando semplicemente di controlla ad uno ad uno se è stato assegnato un valore ai parametri; in tale caso si appende al comando da lanciare. Uno volta controllati tutti i parametri si appende la parola `up` (che indica che l'interfaccia va abilitata) e si lancia il comando in una `check_call` (nessun output generato) controllando che non si sia verificata un'eccezione.
Se tutto è andato a buon fine si chiama la funzione di successo `command_success` passandogli l'object id del documento mongo creato.

Return Il dizionario di successo della funzione `command_success` col campo `data` a `None` e il campo `logid` contenente l'object id del documento mongo creato e contenente le specifiche dell'operazione appena effettuata.

3.4.5 createalias

```
def createalias( aliasname, address, netmask="", broadcast="" ):

    iface = re.sub(':.*', '', aliasname)

    aliases = ifacestat( namesonly=True )
    if aliases['returncode'] is 0:
        if not iface in aliases['data']:
            return command_error( returncode=197, stderr="No interface
                found with such name: " + iface )
    else:
        return aliases

    return ifaceup( iface=aliasname, address=address, netmask=netmask,
        broadcast=broadcast )
```

Nasce per consentire la creazione di una interfaccia alias dal pannello web di nomodo. Utilizza la funzione ifaceup.

Parametri Accetta gli stessi parametri di `ifaceup`, che sono i seguenti:

- `aliasname`: È il nome della nuova interfaccia alias da creare. Può essere ricavata usando la funzione getnewifacealiasname
- `address`: L'indirizzo della nuova interfaccia. È l'unico parametro obbligatorio insieme al nome dell'alias
- `netmask=""`: La maschera di rete per la nuova interfaccia. Non è obbligatorio e se non passato viene calcolata dal sistema
- `broadcast=""`: L'indirizzo di broadcast su cui l'interfaccia invia i pacchetti destinati a tutti gli host della rete. Non è obbligatorio e se non passato viene calcolato dal sistema

Funzionamento Il nome dell'interfaccia da creare può essere ottenuto chiamando la funzione getnewifacealiasname ma è comunque data la possibilità all'utente di scegliere un nome personalizzato. A causa di ciò a differenza delle altre funzioni in `createalias` bisogna verificare almeno se l'interfaccia di cui si vuole creare l'alias esiste.
Viene quindi estratta dal parametro `aliasname` il nome dell'interfaccia principale, eliminando il due punti e l'identificativo dell'alias, usando la libreria per le espressioni regolari `re`. Ad esempio se `aliasname` è `ens1:0` si elimina `:0` ottenendo `ens1`.
Dopodichè si chiama la funzione ifacestat col parametro `namesonly` settato a `True` per ottenere i nomi di tutte le interfacce del sistema.
Viene ovviamente verificato che le operazioni della funzione chiamata non abbiano generato errori verificare la chiave `returncode` del dizionario ottenuto da tale funzione; se `returncode` è diverso da 0 viene restituito il dizionario di errore così come è stato ottenuto al chiamante. Se invece l'operazione

è andata a buon fine si verifica che l'interfaccia ottenuta dal precedente `re.sub` sia presente nella lista di interfacce del sistema. Se non presenta viene generato un errore personalizzato chiamando la funzione `command_error`, mentre se presente semplicemente viene chiamata la funzione precedentemente discussa ifaceup che passandogli paripari i parametri inseriti dalll'utente. Con la sua sintassi questa funzione riesce anche a creare una interfaccia alias.

Return Restistuisce il dizionario di ritorno generato dalla funzione ifaceup.

3.4.6 destroyalias

```
def destroyalias( aliasname ):
    return ifacedown( aliasname )
```

È l'inverso della funzione createalias e serve per rimuovere una interfaccia alias dal sistema.

Parametri Accetta un solo parametro `aliasname` che è il nome dell'interfaccia alias da distruggere.

Funzionamento A differenza della funzione createalias qui non c'è bisogno di verificare se l'interfaccia esista in quanto all'utente non è data la possibilità di definire una interfaccia alias personalizzata da distruggere ma semmplicemente ne sceglie una da una lista di intrefacce presenti nel sistema.
Viene quindi semplicemente chiamata la funzione ifacedown passandogli il nome dell'interfaccia.

Return Restituisce il dizionario di ritorno generato dalla funzione ifacedown.

3.4.7 editiface

```
def editiface( iface, address="", netmask="", broadcast="" ):
    return ifaceup( iface=iface, address=address, netmask=netmask,
        broadcast=broadcast )
```

La funzione consente di modificare dei parametri dell'interfaccia, quali indirizzo, maschera di rete e indirizzo di broadcast.

Parametri Prende fino a 4 parametri:

- `iface`: è il nome dell'interfaccia da modificare
- `address=""`: non è obbligatorio e se passato indica il nuovo indirizzo che l'interfaccia `iface` deve avere
- `netmask=""`: non è obbligatorio e se passato indica la nuova maschera di rete che l'interfaccia `iface` deve avere
- `broadcast=""`: non è obbligatorio e se passato indica il nuovo indirizzo di broadcast che l'interfaccia `iface` deve avere

Funzionamento Non è obbligatorio verificare se l'interfaccia `iface` è presente nel sistema (così come fa createalias) in quanto l'utente non ha la possibilità di specificare a mano l'interfaccia da modificare ma semplicemente la sceglie da una lista.
Semplicemenete chiama la funzione ifaceup passandogli tutti i parametri che ha ricevuto; a causa della sua sintassi generica questa funzione consente anche la modifica di interfacce.

Return Restituisce il dizionario di ritorno generato dalla funzione ifaceup.

3.4.8 getroutes

```
def getroutes():

    command = ['route', '-n']

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )


    output.pop(0)
    header = output.pop(0).split()
    routes = list( map( lambda route: dict(zip(header, route.split())),
        output ) )


    return command_success( data=routes )
```

La funzione serve ad ottenere la lista di tutte le rotte utilizzate dal sistema per uscire sulla rete.

Parametri Non prende parametri.

Funzionamento

```
    command = ['route', '-n']

    try:
        output = check_output(command, stderr=PIPE, universal_newlines=
            True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )
```

Nella prima parte del codice viene costruito il comando da lanciare per ottenere le rotte. Si è usato il comando `route` passandogli il parametro `-n` in modo da ottenere la lista delle rotte senza che gli indirizzi di rete vengano risolti.
Una voilta costruito il comando viene lanciato usando la funzione `check_output` che oltre ad eseguirlo restituisce anche l'output dello stesso. In caso venga generata l'eccezione `CalledProcessError` questa viene catturata e passata al comando command_error che crea un dizionario di errore che viene restituito all'utente.

```
    output.pop(0)
    header = output.pop(0).split()
    routes = list( map( lambda route: dict(zip(header, route.split())),
        output ) )

    return command_success( data=routes )
```

Nella seconda parte prima di tutto viene eliminato la prima riga del comando che solitamente contiene la stringa *Kernel IP routing table* (a noi non utile) usando la funzione `pop()`.
Viene poi rimosso dall'output e splittato l'header; questo header contiene la descrizione breve dei campi della tabella di routing, e solitamente è il seguente:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use
   Iface
```

Nell'ultima parte usando la funzione `map()` viene creata una lista di dizionari in cui:

- Ogni elemento delle lista contiene un dizionario con le specifiche di una rotta
- Ogni dizionario ha tanti campi quanti ne ha l'header, e ogni campo contine come chiave un elemento dell'header e come valore il valore della rotta inerente a quell'elemento header

Un esempio vale più di mille parole, quindi ecco un esempio del dizionario generato su un sistema Ubuntu Server 16.04 che gira su un container LXD:

```
[{'Destination': '0.0.0.0',
  'Flags': 'UG',
  'Gateway': '10.100.10.1',
  'Genmask': '0.0.0.0',
  'Iface': 'eth0',
  'Metric': '0',
  'Ref': '0',
  'Use': '0'},
{'Destination': '10.0.0.0',
  'Flags': 'U',
  'Gateway': '0.0.0.0',
  'Genmask': '255.0.0.0',
  'Iface': 'eth0',
  'Metric': '0',
  'Ref': '0',
  'Use': '0'},
{'Destination': '10.100.10.0',
  'Flags': 'U',
  'Gateway': '0.0.0.0',
  'Genmask': '255.255.255.0',
  'Iface': 'eth0',
  'Metric': '0',
  'Ref': '0',
  'Use': '0'}]
```

Alla fine viene chiamata la funzione di successo `command_success` passandogli il dizionario delle rotte nel campo data.

Return Restituisce il dizionario di successo, con il campo `data` contenente il dizionario delle rotte descritto nel paragrafo **funzionamento**.

3.4.9 addroute

```
def addroute(gw, net, netmask, default=False):

    logid = mongolog( locals() )

    command = ['route', 'add']
    if default:
        command = command + ['default', 'gw', gw]
    elif net is None or netmask is None:
        command_error( returncode=201, stderr='On non-default route you
            must enter "net" and "netmask" parameters' )
    else:
        command = command + ['-net', net, 'netmask', netmask, 'gw', gw]

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

La funzione nasce per aggiungere una rotta alle rotte già presenti nel sistema, visualizzabili chiamando la funzione getroutes.

Parametri Accetta 4 parametri di cui la maggior parte obbligatori:

- `gw`: è l'indirizzo su cui uscire sulla rete `net`
- `net`: rappresenta la rete a di destinazione a cui la rotta fa riferimento
- `netmask`: è la maschera di rete della rete di destinazione, identificata dal parametro `net`
- `default=False`: se `True` indica di creare un default gateway

Funzionamento La maggior parte della funzione si basa sulla costruzione del comando. Le prime due keyword sono predenfinite e sono `route add` che indicano la creazione di una nuova rotta. Entrando nell'`if` distinguiamo tre situazioni:

- Il parametro `default` è `True`: si vuole creare un default gateway, si ha quindi solo bisogno del gateway identificato dal parametro `gw`. Il comando viene costruito inserendo questo unico parametro
- `default` è `False` ma i parametri sono `None` o stringhe vuote: c'è stato un errore nella chiamata, se `default` è `False` per costruire la rotta c'è bisogno di tutti e tre i parametri `gw`, `net` e `netmask`
- `default` è `False` e si hanno tutti i parametri: si vuole creare una nuova rotta, e il comando viene costruito inserendo tutti questi parametri

Dopo la costruzione del comando lo si lancia usando la `check_call` (in quanto non viene restituito output) e si chiama la funzione di errore o di successo a secondo del risultato. Alla funzione di successo viene passato l'object id del documento mongo creato.

Return Restituisce il dizionario di successo con la chiave `data` a `None` (nessun output restituito) e la chiave `logid` contenente l'object id del mongolog creato contenente le informazioni sull'operazione eseguita.

3.4.10 defaultroute

```
def defaultroute(gw): return addroute(gw, net=None, netmask=None, default
    =True)
```

La funzione è definita come *Funzione alias* in quanto basata quasi completamente su un'altar funzione. Prende in carico di creare un default gateway chiamando la funzione addroute con i giusti parametri.

Parametri Prende un solo parametro che è il gateway da assegnare.

Funzionamento Semplicemente chiamta la funzione addroute passandogli il gateway nel parametro `gw`, settando la variabile `default` a `True` e le altre variabili a `None`.

Return Restituisce il dizionario di ritorno della funzione addroute.

3.4.11 delroute

```
def delroute(route):

    logid = mongolog( locals() )

    if not type(route) is type(dict()):
        command_error( returncode=202, stderr='delroute function can only
            accept a dictionary as argument', logid=logid )

    command = [ 'route', 'del', '-net', route['Destination'], 'netmask',
        route['Genmask'], 'gw', route['Gateway'] ]

    try:
        check_call(command)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )
```

La funzione nasce per cancellare una delle rotte presenti nel sistema.
ATTENZIONE: leggere attentamente il paragrafo *Prametri* prima di utilizzarla, in quanto è l'unica che si comporta direttamente sulla questione "parametri in ingresso".

Parametri Accetta un solo parametro che è un dizionario contenente le specifiche della rotta da eliminare. Questo dizionario deve essere lo stesso che si ottiene chiamando la funzione getroutes. Ad esempio prendendo come riferimento l'esempio in sezione 2.4.8 al paragrafo *Funzionamento* volendo cancellare l'ultima rotta dovrei prendere il seguente dizionario (che è il terzo) e passarlo per intero al questa funzione:

```
{'Destination': '10.100.10.0',
  'Flags': 'U',
  'Gateway': '0.0.0.0',
  'Genmask': '255.255.255.0',
  'Iface': 'eth0',
  'Metric': '0',
  'Ref': '0',
  'Use': '0'}]
```

Funzionamento L'opreazione è classificata "sensibile" viene quindi creato un mongolog contenente le informazioni sull'operazione. Viene poi controllato il tipo di parametro in ingresso, e se non è un dizionario viene generato un errore personalizzato usando la funzione command_error.
La costruzione del comando avviene come si vede in codice, mixando le direttive alle informazioni contenute nel dizionario in input. Infine viene lanciato il comando, verificato che non sono successe eccezioni e se tutto è andato a buon fine viene chiamata la funzione di successo passandogli l'object id del documento mongo creato.

Return Restituisce il dizionario di successo senza dati (`data=None`) ma con l'object id del mongolog creato alla chiave `logid`.

3.5 Cron

Il cron è un software incluso in tutte le distribuzioni Linux che copre il compito di eseguire detereminati task (definiti dall'utente) con una certa frequenza e a specifici giorni e orari.
Un esempio di task ricorrente può essere ad esempio l'aggiornamento del database dei file per la ricerca indicizzata.
Il cron di Linux può essere di sistema o dell'utente. Siccome usando il cron di sistema è possibile lanciare i processi utilizzando qualsiasi utente del sistema si è deciso di tralasciare il cron utente ed implementare in nomodo solo quello di sistema.
Distinguiamo 2 tipi di cron:

- Il cron utilizzato da `cronspath` e dagli script in `cron.d` che sono veri e propri script di cron, contenente cioè tutte le informazioni sulla tempistica di lancio del comando o dello script, l'utente e il comando stesso. Il comando in questo caso deve essere contenuto in una sola riga. In nomodo la creazione di tali script è possibile chiamando la funzione addcron
- Il cron delle cartelle `/etc/cron.daily`, `/etc/cron.hourly`, `/etc/cron.monthly` e `/etc/cron.weekly` che sono invece degli script o dei programmi (solitamente scrcitti in bash) che vengono eseguiti uno alla volta ordinati per nome all'ora e alla data definiti di default nel file `/ect/crontab`. In nomodo la creazione di tali script avviene tramite la chiamata alle funzioni derivate da adddefaultcron, ad es. `adddailycron`

Da notare che le funzioni per la modifica, la rimozione o il rename dei cron non sono presenti in quanto vengono usate le funzioni della libreria utilities per la gestione dei file quali writefile o filedel.

3.5.1 listcrontabs

```
def listcrontabs():

    basedir = '/etc/'
    paths = ['cron.d', 'cron.daily', 'cron.hourly', 'cron.monthly', 'cron
       .weekly']

    cronlist = dict()
    for path in paths:
        flist = os.listdir(basedir + path)
        flist.remove('.placeholder')
        cronlist.update({ path: flist })

    return command_success( data=cronlist )
```

La funzione restituisce la lista di tutti i crontab installati nel sistema.

Parametri Non prende nessun parametro.

Funzionamento Definisce due variabili, `basedir` che è dove sono presenti le cartelle del cron e `paths` che è una lista contenente i nomi delle cartelle del cron. Notare che la lista delle cartelle non contiene il file `crontabs` che si è deciso di non utilizzare in quanto egregiamente sostituito dalla cartella `cron.d`.
Entrando nel ciclo for costruisce il dizionario `cronslist` la cui chiave è il nome della cartella mentre il valore è la lista dei file presenti nella cartella stessa, che sarebbero quindi la lista dei crontab. Da questa lista viene eliminato il file `.placeholder` che è solo un file richiesto dal sistema e non utile all'utente finale.

Return Restitusice il dizionario di successo contenente nella variabile `data` il dizionario creato dalla funzione come spiegato nel funzionamento. Un esempio del dizionario è il seguente:

3.5.2 getcronname

```
def getcronname(): return 'nomodo-' + datetime.datetime.now().strftime('%
   Y%m%d%H%M%S')
```

Genera un nome da usare per un nuovo cron e funziona allo stesso modo di getreponame. Viene usata principalmente in due fasi:

1. Viene chiamato dal frontend per riempire il campo *name* all'aggiunta di un nuovo crontab. L'utente ovviamente ha la possibiltà di cammbiare questo nome prima dell'aggiunta dello script

2. Chiamato dalle funzione di aggiunta cron addcron e adddefaultcron in quanto se il parametro `name` non è stato passato vanno ad utilizzare il nome restituito da questa funzione. Questa operazione sembra ridondante in quanto già chiamata dal frontend come spiegato al punto 1, ma serve a dare solidità al codice

Parametri Non prende nessun parametro.

Funzionamento Semplicemente contatena la stringa "nomodo-" alla data e ora odierna col formato che si vede nel codice.

Return È una delle pochissime funzione che non chiama una delle due funzioni di ritorno di nomodo. Restituisce la stringa costruita come spiegato nel funzionamento.

3.5.3 addcron

```
def addcron( command, name="", user="root", minute='*', hour='*', dom='*'
    , month='*', dow='*' ):

    cronspath = '/etc/cron.d/'

    #New cron gets a random name if user did not provide it
    if not name: name = getcronname()

    logid = mongolog( locals() )

    with open(cronspath + name, 'w') as newcron:
        newcron.write( minute + ' ' + hour + ' ' + dom + ' ' + month + ' '
            ' + dow + ' ' + user + ' ' + command + '\n' )

    return command_success( data=cronspath+name, logid=logid )
```

La funzione nsace per la creazione di uno script di cron vero che si vada ad aggiungere a quelli presenti nella cartella `/etc/cron.d/`. Per "script di cron vero" si intende un file di direttive che contenga la sintassi del cron; come spiegato nell'introduzione alla sezione 2.5 questi si differenziano dagli script specifici in quanto questi ultimi sono veri e propri applicativi, usualmente scritti in bash, mentre gli script di cron contengono anche le informazioni sulla tempistica dell'esecuzione del comando e il comando deve essere contenuto in una linea o deve essere la chiamata ad un applicativo.
La spiegazione di centos in questo esempio chiarisce ancora meglio come deve essere strutturato uno script di cron:ù

```
# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,
    wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name  command to be executed
```

Si intuisce da questo esempio che mentre i campi `dom` e `dow` interferiscano tra di loro. In realtà settando entrambi il comando verrà eseguito sia nel "day of month" sia nel "day of week" specificari.

Parametri Accetta un sacco di parametri, qui spiegati:

- `command`: È il comando che deve essere eseguito. Deve essere di una sola riga e può essere sia un comando di bash che la chiamata ad un'applicativo di sistema, che può essere in qualsiasi linguaggio, incluso bash stesso (ovviamente)

- `name=""`: È il nome del nuovo script di bash. Se non specificato viene gererato automaticamente attraverso la chiamata alla funzione getcronname. In realtà la funzione viene chiamata direttamente dal frontend per settare il nome, ma per la massima sicurezza viene fatto un controllo per vedere se il nome è vuoto, e generarlo in tal caso
- `user="root"`: È l'utente per conto di cui il comando verrà lanciato. Di default è l'utenza amministrativa.
- `minute="*"`: È il minuto in cui il comando verrà eseguito. Di default è * che indica ogni minuto
- `hour="*"`: È l'ora in cui il comando verrà eseguito. Di default è * che indica ogni ora. Accetta valori da 0 a 24 ed è basato sul fuso orario del sistema
- `dom="*"`: È il giorno del mese (dom="day of month") in cui il comando verrà eseguito. Di default è * che indica ogni giorno del mese. Accetta valori da 1 a 31 se il mese ha 31 giorni
- `month="*"`: È il mese in cui il comamndo deve essere eseguito. Di default è * che indica ogni mese. Accetta valori da 1 da 12
- `dow="*"`: È il giorno della settimana in cui il comando deve essere esguito. Di default è * che indica ogni gionro della settimana. Accetta valori da 1 a 7

Funzionamento Una volta settato il path in cui il file verrà creato (`/etc/cron.d/`) si crea il mongolog in quanto bisogna tenere traccia di chi ha creato il file, e semplicemente usando una `with open` si apre il file in scrittura e si scrivono tutti le informazioni ricevute come parametri.

Return Restituisce il percorso del file creato e il logid del documento mongo contenente le informazioni sulla operazione nel dizionario di successo generato dalla funzione `command_success`.

3.5.4 adddefaultcron

```
def adddefaultcron(command, cronspath, name):

    #New cron gets a random name if user did not provide it
    if not name: name=getcronname()

    logid = mongolog( locals() )

    with open(cronspath + name, 'w') as newcron:
        newcron.write( command + '\n' )

    return command_success( data=cronspath+name, logid=logid )


def addhourlycron(command, name=""): return addefaultcron( name=name,
   command=command, cronspath='/etc/cron.hourly/' )
def adddailycron(command, name=""): return addefaultcron( command=command
   , cronspath='/etc/cron.daily/' )
def addweeklycron(command, name=""): return addefaultcron( command=
   command, cronspath='/etc/cron.weekly/' )
```

```
def addmonthlyycron(command, name=""): return addefaultcron( command=
    command, cronspath='/etc/cron.monthly/' )
```

La funzione si contrappone a addcron e a differenza di questo serve a creare uno script nelle cartelle `/etc/cron.hourly`, `/etc/cron.daily`, `/etc/cron.weekly` e `cron.monthly`. Come spiegato gli script di queste cartelle a differenza di quelli in `/etc/cron.d` sono veri e propri script (solitamente script di bash) che vengono eseguiti ordinati per nome e secondo le direttive del file `/etc/crontab`. Qui un esempio di tale file preso da un Ubuntu 16.04 server:

```
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the 'crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts
    --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts
    --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts
    --report /etc/cron.monthly )
```

Quindi ad esempio tutti gli script in `/etc/cron.daily` verranno eseguiti ogni mattina alle ore 6:25.

Parametri Non prendendo informazioni ne sulla tempistica ne sull'utente prende pochi parametri, qui presentati:

- `command`: È il contenuto dello script che verrà aggiunto alle cartelle. In questo caso può essere multilinea. Il fontend può leggere questo contenuto ad esempio da una text area
- `cronspath`: Dato che questa funzione non andrebbe mami richiamata direttamente ma sempre attraverso le 4 funzioni che si vedono nelle ultime 4 righe del codice questo è un parametro che indica la cartella dove lo script sarà posizionato; questo cambia a seconda della funzione chiamata
- `name`: È il nome dello script di cron, che come per addcron se non presente viene generato tramite la chiamata alla funzione getcronname

Funzionamento Come per `addcron` la funzione non fa altro che verificare che il nome sia presente e generarlo in caso negativo, creare un documento mongo che memorizzi l'operazione effettuata e scrivere lo script (contenuto nella variabile `command`) nel nuovo file.
La funzione principale non va mai chiamata direttamente ma bisogna sempre utilizzare le 4 funzioni intermediare che si vedono nelle ultime 4 righe del codice e che vanno a settare correttamente la variabile `cronspath` a seconda della tempistica di esecuzione dello script.

Return Ugualemente a `addcron` restituisce il dizionario di successo contenete il percorso del nuovo file nella variabile `data` e il l'object id del documento mongo contenente le informazioni sull'operazione nella variabile `logid`.

3.6 Sistema

Questa libreria è relativamente piccola per adesso ma sarà riempita con altre funzione in seguito. Contiene tutte quelle funzione che riguardano il sistema in generale, come ad esempio la funzione per la visualizzione e il cambio del nome macchina.

3.6.1 hostname

```
def hostname(newhostname=""):

    command = ['hostname']

    if newhostname:
        logid = mongolog( locals() )
        command.append(newhostname)

    try:
        hostname = check_output(command, stderr=PIPE, universal_newlines=
            True)
    except CalledProcessError as e:
        if newhostname:
            return command_error( e, command, logid )
        else:
            return command_error( e, command )

    if newhostname:
        return command_success( logid=logid )
    else:
        return command_success( data=hostname )
```

La funzione serve sia ad ottenere il nome macchina che a cambiarlo.

Parametri Accetta un solo parametro opzionale `newhostname` che contiene il nuovo nome macchina da assegnare alla stessa.

Funzionamento Si comporta similmente alla funzione `hostname` dei sistemi linux, ossia se chiamata senza alcun parametro restituisce l'hostname della macchina sulla quale viene eseguito, mentre e chiamato con un parametro allora setta il contenuto di questo parametro come nuovo nome macchina. Viene quindi fatto un controllo, se il parametro non obbligatorio `newhostname` contiene una stringa allora crea un nuovo momgolog e assegna alla macchina questa stringa come nome host, restituisce l'hostname della macchina altrimenti.

Return Restituisce il dizionario di successo contenente l'hostname se `newhostname` è vuota, l'object id del mongolog altrimenti.

3.6.2 getsysteminfo

```
def getsysteminfo( getall=True, getproc=False, getcpu=False, getmem=False
    ):

    #Tuple to return
    toreturn = ()

    ##### CPU #####
    if getall or getcpu:

        #Reading cpu stat from /opt files
        with open('/proc/cpuinfo', 'r') as cpuorig:
            cpuraw = cpuorig.read().splitlines()

        #Removing duplicate lines by converting list() to set()
        cpuraw = set(cpuraw)

        #Removing empty lines
        cpuraw = list( filter( None, cpuraw ) )

        #Removing useless lines
        linestoremove = ('flags', 'apicid', 'processor', 'core id', '
            coreid')
        cpuraw = list( filter( lambda line: not any(s in line for s in
            linestoremove), cpuraw ) )

        #Deleting all tabulation and spaces for each line of the cpuraw
            set cpuraw
        cpuraw = map( lambda line: re.sub('[\t| ]*:[\t| ]*', ':', line),
            cpuraw )


        #We got three fields named "cpu Mhz", but to use them as
            dictionry keys
        #we need to rename them all
        cpuaf = list()
        i = 1
        for line in cpuraw:
            #Adds an incremental number to the key
            if 'mhz' in line.lower():
                cpuaf.append( re.sub('^.*:', 'core' + str(i) +' MHz:',
                    line) )
                i += 1
            else: cpuaf.append( line )


        #Buiding final dictionary cotaining cpu information in the right
            form
```

```
        cpu = dict()
        for line in cpuaf:
            line = line.split(':')
            cpu.update({ line[0]: line[1] })

        #Adding cpu dict to the tuple to return
        toreturn = toreturn + (cpu,)



    ##### MEMORY #####
    if getall or getmem:

        #Reading memory status from /opt files
        with open('/proc/meminfo', 'r') as memorig:
            memraw = memorig.read().splitlines()


        #Filling mem dict with memory information
        mem = dict()
        for line in memraw:
            line = re.sub('␣', '', line)                #Removing spaces
                for each line
            line = line.split(':')                      #Splitting by
                colon
            mem.update({ line[0].lower() : line[1] })   #Appending the
                dictionary to a list to return

        toreturn = toreturn + (mem,)




    #### PROCESSES #####
    if getall or getproc:

        #Reading processes status using top command
        command = ['top', '-b', '-n1']

        try:
            procraw = check_output(command, stderr=PIPE,
                universal_newlines=True).splitlines()
        except CalledProcessError as e:
            return command_error( e, command )



        #Removing headers from the output of top command
```

```
        i = 0
        while 'PID' not in procraw[i]: i+=1
        procraw = procraw[i:]
        proc = list(map( lambda line: line.split(), procraw ))
        toreturn = toreturn + (proc,)


    #dict, dict, list
    return command_success( data=toreturn )
```

Nasce per la creazione della dashboard della pagina *system* in nomodo. Serve ad ottenere informazioni riguado memoria, cpu e processi del sistema.

Parametri Accetta tre parametri il cui valore definisce la quantità di informazioni che si desidera ottenere:

- `getall=True`: Questa variabile se True prevale sulle altre, ossia se questa variabile è settata a True non importa come sono settate le altre, la funzione restituiurà sempre tutte le informazioni possibili
- `getproc=False`: Funziona solo se `getall=False`, indica alla funzione di includere nella tupla di ritorno le informazioni sui processi attualmente attivi sul sistema
- `getcpu=False`: Funziona solo se `getall=False`, indica alla funzione di includere nella tupla di ritorno anche le informazioni sulla CPU della macchina
- `getmem=False`: Funziona solo se `getall=False`, indica alla funzione di includere nelle tupla di ritorno anche le informazioni sulla memoria della macchina

Funzionamento

```
    #Tuple to return
    toreturn = ()
```

All'inizio della funzione viene allocata una tupla vuota chiamata `toreturn` che sarà l'unica tupla che verrà restituita all'utente. A seconda dei parametri passati alla funzione questa avrà più o meno elementi. La tupla si è resa obbligatoria in quantosi devono restituire retituire più variabili.

```
    ##### CPU #####
    if getall or getcpu:

        #Reading cpu stat from /opt files
        with open('/proc/cpuinfo', 'r') as cpuorig:
            cpuraw = cpuorig.read().splitlines()

        #Removing duplicate lines by converting list() to set()
        cpuraw = set(cpuraw)

        #Removing empty lines
        cpuraw = list( filter( None, cpuraw ) )

        #Removing useless lines
```

```
        linestoremove = ('flags', 'apicid', 'processor', 'core id', '
            coreid')
        cpuraw = list( filter( lambda line: not any(s in line for s in
            linestoremove), cpuraw ) )

        #Deleting all tabulation and spaces for each line of the cpuraw
            set cpuraw
        cpuraw = map( lambda line: re.sub('[\t| ]*:[\t| ]*', ':', line),
            cpuraw )


        #We got three fields named "cpu Mhz", but to use them as
            dictionry keys
        #we need to rename them all
        cpuaf = list()
        i = 1
        for line in cpuraw:
            #Adds an incremental number to the key
            if 'mhz' in line.lower():
                cpuaf.append( re.sub('^.*:', 'core' + str(i) +' MHz:',
                    line) )
                i += 1
            else: cpuaf.append( line )


        #Buiding final dictionary cotaining cpu information in the right
            form
        cpu = dict()
        for line in cpuaf:
            line = line.split(':')
            cpu.update({ line[0]: line[1] })

        #Adding cpu dict to the tuple to return
        toreturn = toreturn + (cpu,)
```

La prima parte della funzione è quella che ricava informazioni sulla CPU del sistema. Come si vede dal primo `if` se la variabile `getall` è vera allora la variabile `getcpu` non viene affatto controllata, e si deduce che l'utente quindi voglia anche informazioni sulla CPU.
Le informazioni sono ricavate dal file `/proc/cpuinfo` che come sappiamo facente parte del filesystem `proc` le informazioni al suo interno vengono calcolate al momento dell'apertura e quindi non è un vero file. Dopo l'apertura (che restituisce una lista di stringhe a causa dell'utilizzo della funzione `splitlines()`) si effettuano alcuni accorgimenti sull'output, specialmente perchè questo andrà a costruire un dizionario chiave-valore. In particolare:

- Si eliminao le righe simili. I processori moderni hanno come minimo 2 core per CPU; questo implica una duplicazione di informazioni restituite all'utente all'apertura di questo file. Si trasforma quindi la lista in un set in quanto un set non può avere valori duplocati, e vengono quindi rimossi automaticamente alla conversione

- Vengono eliminate le righe vuota usando la funzione `filter()` e dandogli come funzione lambda

la keywork `None`

- Vengono eliminate le righe non utili per il frontend che sono *flags, apicid, processor, core id* e *coreid*
- Tramite la funzione `map` si scorrono tutte le righe eliminando gli spazi o le tabulazioni tra la chiave e il valore della riga. Ad esempio `cpucores : 2` diventa `cpucores:2` così da agevolare la creazione del dizionario che andrà restituito
- Il tipo `set` elimina le righe uguali ma non le righe in cui solo le chiavi sono uguali. Inoltre per ogni core della CPU abbiamo bisogno di sapere la sua frequenza, ma come sappiamo per la costruzione del dizionario non possono esserci chiavi uguali. Si entra quindi nel primo ciclo for dove per ogni riga del `set` se la riga contiene la stringa *mhz* (ossia è una riga indicante la frequenza del core) rinomina l'attuale chiave da *cpu MHZ* a *core#Mhz* dove # è un numero incrementale indicato dalla variabile `i` nella funzione. Avremo così una chiave univoca per ogni core. [6]

Viene infine costruito il dizionario da inserire nella tupla `toreturn`. Ogni riga del set viene splittata per due punti ottenendo una lista di due valori dove il primo elemento è la chiave e il secondo è il valore. Vengono quindi inseriti nel dizionario `cpu` e questo stesso dizionario infine lui stesso inserito nella tupla `toreturn`.

```
    if getall or getmem:

        #Reading memory status from /opt files
        with open('/proc/meminfo', 'r') as memorig:
            memraw = memorig.read().splitlines()


        #Filling mem dict with memory information
        mem = dict()
        for line in memraw:
            line = re.sub(' ', '', line)                #Removing spaces
                for each line
            line = line.split(':')                      #Splitting by
                colon
            mem.update({ line[0].lower() : line[1] })   #Appending the
                dictionary to a list to return

        toreturn = toreturn + (mem,)
```

La seconda parte del codice riguarda le informazioni sulla memoria della macchina, e richiede meno elaborazione delle informazioni sulla CPU. Come prima se `getall` è `True` si entra nel codice senza nemmeno considerare la variabile `getmem`.
Questa volta il file del filesystem `proc` che contiene le informazioni sulla memoria è `/proc/meminfo`. Viene quindi aperto, letto il contenuto, diviso per righe ed inserito nella variabile `memraw`, che diventa quindi una lista di strighe.
Le informazioni non contengono righe uguali, righe vuote ecc. quindi le uniche operazioni da compiere sono eliminare gli spazi vuoti dalle righe (utilizzando `re.sub`), dividere le linee per due punti, costruire il dizionario dove inserire queste righe e inserire lo stesso nella tupla `toreturn`.

[6] Notare che oltre all'aggiunta del numero incrementale viene anche rimosso lo spazio

```
    #### PROCESSES #####
    if getall or getproc:

        #Reading processes status using top command
        command = ['top', '-b', '-n1']

        try:
            procraw = check_output(command, stderr=PIPE,
                universal_newlines=True).splitlines()
        except CalledProcessError as e:
            return command_error( e, command )



        #Removing headers from the output of top command
        i = 0
        while 'PID' not in procraw[i]: i+=1
        procraw = procraw[i:]
        proc = list(map( lambda line: line.split(), procraw ))
        toreturn = toreturn + (proc,)
```

La terza ed ultima parte del codice ricava le informazioni sui processi attualmente in esecuzione sulla macchina. In questo caso non c'è un file (o almeno non ciè un file chiaro ed utile al nostro scopo) che contenga tutte le informazioni che a noi servono. Dopo essere quindi entrati nell'if (`getall=True` o `getproc=True`) viene lanciato l'applicativo `top` con i parametri `b` e `-n1` che servono rispettivamente a lanciarlo in maniera non interattiva e a restituire la lista dei processi una sola volta. L'output viene diviso per righe (`splitlines()`) ed inserire in `procraw` che diventa una lista di stringhe.
Nell'output del comando è incluso anche l'header che contiene diverse informazioni, ad esempio sui processi in IO wait. Non essendo utili in questa fase del programma cerchiamo di eliminarlli. Si entra quindi in un ciclo `while` iterando sulle righe una alla volta ed incrementando un contatore `i` ad ogni iterazione. L'iterazione si ferma quando si arriva ad una riga in cui sia presente la stringa `PID`, che indica l'inizio della porzione di informazioni che deve essere mantenuta. Sappiamo quindi che le righe nel range 0 - i-1 devono essere eliminate. Viene quindi lanciato il comando `procraw = procraw[i:]` che effettua proprio questa operazione.
Abbiamo quindi ottenuto le informazioni che ci servono, usando quindi la funzione `map` splittiamo per spazio vuoto ottenendo una lista di liste (`proc`) che inseriamo nella tupla `toreturn`.

```
    return command_success( data=toreturn )
```

Non ci rimane quindi che restituire la tupla `toreturn` che contiene le informazioni chieste dall'utente.

Return Restituisce il dizionario di successo contenente alla vatiabile `data` la tupla `toreturn` contenente la qunatità di inormazioni chieste dall'utente, ossia `cpu` (dizionario), `mem` (dizionario) e `proc` (lista di liste) se `getall=True`, le informazioni richieste utilizzando i parametri altrimenti.

3.7 Apache

La libreria *apache* è utile alla gestione del web server Apache e dei suoi componenti, ossia configurazioni (abbreviato in *conf*) e moduli (abbreviato in *mods*).

Le funzioni principali di questa libreria non vengono mai chiamate direttamente ma sempre tramite una funzione alias che è possibile trovare alla fine del codice di ogni funzione.

3.7.1 getobjs

```
#NOTE: Must not be called directly

apacheconfdir = "/etc/apache2/"

def getobjs(objtype):

    availabledir = objtype + '-available/'
    enableddir = objtype + '-enabled/'

    objs = list()

    #Getting enabled vhosts and appending to vhosts list as dictionary
    enabled = set( os.listdir(apacheconfdir + enableddir ) )
    for obj in enabled:
        objs.append({ 'filename': obj, 'active': 1 })

    #Gets nonactive vhosts and appending to vhosts list as dictionary
    notactive = set( os.listdir(apacheconfdir  + availabledir) ).
       difference(enabled)
    for obj in notactive:
        objs.append({ 'filename': obj, 'active': 0 })



    ##### ONLY FOR VHOSTS #####
    #Gathering object information only if objstype == "site"
    if objtype is "sites":
        #Gathering vhosts information to fill the list
        for obj in objs:

            #"vhostcontent" mantains all vhost file content
            with open(apacheconfdir + availabledir + obj['filename']) as
               opened:
                vhostcontent = opened.read().splitlines()


            i = 0
            for line in vhostcontent:
                line = line.lstrip()
                linestosearch = ('Alias', 'DocumentRoot', 'ServerName', '
                   ServerAlias')

                #If any of this words in vhost file add the entire
                   splitted line to the vhost dict
```

```
                if any( line.startswith(s) for s in linestosearch ):

                    #A vhost can handle multiple ServerAlias but dict()
                       cannot accept multiple key with the same string
                    #so we're going to add an incremental number to the
                       key "ServerAlias"
                    if 'ServerAlias' in  line: line = re.sub('ServerAlias
                       ', 'ServerAlias' + str(i), line)
                    i += 1

                    line = line.split(None, maxsplit=1)

                    #vhost dict is a pointer to the original dict in
                       vhosts list, hence an update here means an update
                       to the original dict
                    obj.update({ line[0]: line[1] })


    return command_success( data=objs )

def getvhosts(): return getobjs('sites')
def getmods(): return getobjs('mods')
def getconf(): return getobjs('conf')
```

La funzione dato un tipo di oggetto in input restituisce la lista di tutti gli oggetti di quel tipo presenti in apache. Non viene mai chiamata direttamente ma sempre tramite le tre funzioni che si vedono sulle ultime tre righe del codice. Si deduce che gli oggetti che è possibile ottenere sono i *virtual hosts* ossia i sites che apache gestisce, i moduli ossia i *mods* e le configurazioni, ossia i *conf*.
Notare che la variabile `apacheconfdir` è posizionata all'esterno e quindi globale, in quanto deve essere universale a tutti e anche al fontend che la chiama all'appello.

Parametri La funzione principale accetta un solo parametro che è il tipo di oggetto su sui la funzione lavora, ma visto che questa non viene mai richiamata direttamente le tre funzioni non accettano alcun parametro.

Funzionamento

```
    availabledir = objtype + '-available/'
    enableddir = objtype + '-enabled/'

    objs = list()

    #Getting enabled vhosts and appending to vhosts list as dictionary
    enabled = set( os.listdir(apacheconfdir + enableddir ) )
    for obj in enabled:
        objs.append({ 'filename': obj, 'active': 1 })

    #Gets nonactive vhosts and appending to vhosts list as dictionary
    notactive = set( os.listdir(apacheconfdir  + availabledir) ).
       difference(enabled)
```

```
    for obj in notactive:
        objs.append({ 'filename': obj, 'active': 0 })
```

Nella prima parte del codice innanzitutto vengono definite tre variabili:

- `apacheconfdir`: indica il path sul sistema dove sono presenti le configurazioni e gli oggetti di apache
- `availabledir`: dato in input l'oggetto che si desidera questa variabile indica il path dove sono presenti tutti i file per quell'oggetto, che siano attivati o meno. Ad esempio per i vhosts questo path è `apachecondir + 'sites-available'` mentre per i mods è `apacheconfdir + 'mods-available'`
- `enableddir`: è il path sul sistema dove sono presenti i file di configurazione degli oggetti che sono attivati su apache. Queste cartelle contengono link simbolici ai file nelle cartelle di `availabledir`. Se il link simbolico è presente allora significa che il file dell'oggetto in questione è attivo. Ad esempio nell'installazione di base apache contiene un link simbolico `/etc/apache2/sites-enabled/default-ssl` che punta allo file nella cartella `/etc/apache2/sites-availabl`
  Il fatto che esista questo link nella cartella indica che questo oggetto è attivo per apache, ossia che il sito (o vhost) *default-ssl* è attivo

La variabile da restituire è `objs` che sarà costruita come lista di dizionari, dove ogni dizionario conterrà le informazioni su ognuno degli oggetti di apache di tipo *¡objtype¿*. Per ogni file di un oggetto di apache si è deciso di restituire in output oltre al nome anche un flag che indica che la configurazione è attiva se è 1, disattiva se 0. Si crea quindi prima di tutto una lista di oggetti abilitati, chiamando un `listdir` sulla cartella *enabled* dell'oggetto voluto (ad esempio *sites-enabled*). Si aggiunge quindi ognuno di questi oggetti alla lista di dizionari `objs` assegnandosi anche il flag `active` col valore 1, ad indicare che l'oggetto è attivo.
Dato che i file nella cartella `<objtype>-enabled` sono un sottoinsieme dei file nella cartella `<objtype>-available` per ottenere gli oggetti di *objtype* non attivi bisogna sottrarre l'insieme dei file di `<objtype>-enabled` all'insieme dei file di `<objtype>-available`. Si ottiene quindi una lista degli oggetti di tipo *objtype* che non sono attivi, e li si inserisce nella lista di dizionari `objs` con la flag `active` a 0 ad indicare che tali oggetti non sono attivi.

```
    ##### ONLY FOR VHOSTS #####
    #Gathering object information only if objstype == "site"
    if objtype is "sites":
        #Gathering vhosts information to fill the list
        for obj in objs:

            #"vhostcontent" mantains all vhost file content
            with open(apacheconfdir + availabledir + obj['filename']) as
                opened:
                vhostcontent = opened.read().splitlines()


            i = 0
            for line in vhostcontent:
                line = line.lstrip()
                linestosearch = ('Alias', 'DocumentRoot', 'ServerName', '
                    ServerAlias')
```

```
                #If any of this words in vhost file add the entire
                    splitted line to the vhost dict
                if any( line.startswith(s) for s in linestosearch ):

                    #A vhost can handle multiple ServerAlias but dict()
                        cannot accept multiple key with the same string
                    #so we're going to add an incremental number to the
                        key "ServerAlias"
                    if 'ServerAlias' in  line: line = re.sub('ServerAlias
                        ', 'ServerAlias' + str(i), line)
                    i += 1

                    line = line.split(None, maxsplit=1)

                    #vhost dict is a pointer to the original dict in
                        vhosts list, hence an update here means an update
                        to the original dict
                    obj.update({ line[0]: line[1] })


    return command_success( data=objs )
```

Questo pezzo di codice viene eseguito solo se il tipo di oggetto su cui si sta lavorando è un *site*, cioè un *vhost*, in quanto tutti i site hanno delle informazioni comuni che è possibile inserire nel dizionario di ritorno.
Quindi prima di tutto si cicla su ogni site di apache, aprendo il contenuto, leggendolo e dividendolo per linee, e inserendolo quindi nella variabile `vhostcontent`. Si comincia poi a ciclare sulle linee; per ogni linea vengono innanzitutto rimossi gli spazi iniziali usando la funzione `lstrip()` in quanto molte linee presentano probabilmente una tabulazione iniziale. Viene creata una lista `linestosearch` contenente tutte le direttive la cui riga si vuole estrarre dal file. Se la riga attuale inizia con una di queste direttive (`startswith()`) la riga può ovviamente essere inserita nel dizionario che si andrà a restituire. Come succedeva con la funzione getsysteminfo è possibile che ci siano più direttive con il nome `ServerAlias` e sarebbe quindi impossibile inserirle nel dizionario come chiave; viene quindi lanciato l'ultimo `if` che all'incontro con una di queste direttive aggiunge un valore numerico incrementale che rende la sua definizione unica. Siamo quindi pronti ad inserire queste informazioni nel dizionario. splittiamo quindi la riga per spazio vuoti (`split(None)`) e una sola volta (`maxsplit=1` in quanto potrebbero esserci più spazi vuoti nella riga). Alla fine questi valori vengono inseriti nella lista di dizionari che viene restituita `objs` e viene chiamata la funzione `command_success()` passandogli proprio questa lista e che sancisce la fine della funzione.

Return Restituisce il dizionario di successo se tutte le operazioni sono andate a buon fine, contenente alla chiave `data` la lista di dizionari `objs` in cui ogni dizionario è un oggetto di tipo *objtype* contenente le informazioni sullo stesso. Queste informazioni:

- Se si sta richiedendo info su conf e mods contiene solo il nome e il flag `active` a 0 o 1 che indica se l'oggetto è attivo o meno in apache
- Se si sta richiedendo info sui vhost contiene oltre al nome e il flag `active` anche (se presenti) i valori delle direttive *'Alias', 'DocumentRoot', 'ServerName'* e *'ServerAlias'*

3.7.2 manageapache

```
#NOTE: Must not be called directly
def manageapache(op):

    #Can only accept these parameters
    acceptedparams = ['stop', 'status', 'reload', 'restart']
    if not any(op in param for param in acceptedparams):
        return command_error( returncode=-1, stderr='Bad parameter: '+op
            )
    else:
        command = ['systemctl', op, 'apache2', '-q', '-n0', '--no-pager']


    toreturn = None
    try:
        if op is "status":
            #Avoid to print journal (log) lines in output
            command.append('-n0')

            #We are using Popen here because check_output fails to return
                 stdout and stderr on exit code != 0
            toreturn = Popen(command, stdout=PIPE, universal_newlines=
                True).communicate()[0].splitlines()

            #Filtering useless lines
            linestomantain = ['loaded', 'active', 'memory', 'cpu']
            toreturn = list( filter( lambda line: any(s in line.lower()
                for s in linestomantain), toreturn ) )
            #Formatting output
            toreturn = dict( map(lambda line: line.lstrip().split(':',
                maxsplit=1), toreturn ) )

        else:
            #Logging operation to MongoDB; in this specific case "
                toreturn" contains mongo logid
            toreturn = mongolog( locals() )
            check_call(command)
    except CalledProcessError:
        pass


    return command_success( data=toreturn )

def apachestart(): return manageapache(op='start')
def apachestop(): return manageapache(op='stop')
def apacherestart(): return manageapache(op='restart')
def apachereload(): return manageapache(op='reload')
def apachestatus(): return manageapache(op="status")
```

La funzione nasce per gestire il demone di apache, cioè per avviarlo, fermarlo ecc. Come prima la funzione principale non va mai chiamata direttamente ma sempre tramite gli alias che si leggono nelle ultime cinque righe del codice.

Parametri La funzione principale prende un solo parametro che è l'indicazione sull'operazione da eseguire, ma visto che questa non viene mai chiamata direttamente la trascuriamo.
Le funzioni alias invece non prendono alcun argomento, in quanto la loro chiamata contiene già è sufficiente alla libreria per capire cosa il frontend vuole.

Funzionamento

```
    #Can only accept these parameters
    acceptedparams = ['stop', 'status', 'reload', 'restart']
    if not any(op in param for param in acceptedparams):
        return command_error( returncode=-1, stderr='Bad parameter: '+op
            )
    else:
        command = ['systemctl', op, 'apache2']

    toreturn = None
```

Anche se la funzione non deve essere chiamata direttamente è sempre buono effettuare un controllo per verificare se il valore del parametro `op` rientra tra quelli possibili. Se non è così viene chiamata la funzione di errore `command_error` generando un errore custom, altrimenti viene composto il comando che si andrà ad eseguire ed inserito nella variabile `command`. Viene poi dichiarata la variabile `toreturn` senza un tipo, in quanto questo cambierà a seconda del parametro `op`.

```
    try:
        if op is "status":
            #Avoid to print journal (log) lines in output
            command.append('-n0')

            #We are using Popen here because check_output fails to return
                 stdout and stderr on exit code != 0
            toreturn = Popen(command, stdout=PIPE, universal_newlines=
                True).communicate()[0].splitlines()

            #Filtering useless lines
            linestomantain = ['loaded', 'active', 'memory', 'cpu']
            toreturn = list( filter( lambda line: any(s in line.lower()
                for s in linestomantain), toreturn ) )
            #Formatting output
            toreturn = dict( map(lambda line: line.lstrip().split(':',
                maxsplit=1), toreturn ) )

        else:
            #Logging operation to MongoDB; in this specific case "
                toreturn" contains mongo logid
            toreturn = mongolog( locals() )
            check_call(command)
```

```
    except CalledProcessError:
        pass


    return command_success( data=toreturn )
```

C'è una grande differenza nel chiamare la funzione con `op='status'` e con `op` uguale a qualsiasi altra cosa, in quanto *status* genera un output che deve essere restituito mentre le altre no.
Si va quindi prima di tutto a verificare il valore di `op`:

- Se chiamata con tutto al di fuori di *'status'* viene semplicemente creato un mongolog, lanciato il comando ed inserito l'object id del documento mongo nel dizionario di ritorno tramite la funzione `command_success`
- Se invece viene chiamata con `op='status'` vengono fatte giusto un pò di operazioni in più. Prima di tutto viene aggiunto il comando l'argomento `-n0` che omette la stampa del giornale di systemd riguardante l'unità stessa, ossia *apache.service*.
  Viene poi lanciato il comando. Notare che il lancio avviene con `Popen` in quanto a parte che non ci interessa niente del codice di ritorno, ma inoltre questo è diverso da 0 se ad esempio il servizio è fermo, cosa che fa andare l'applicativo in crash.
  Viene quidi cerata una lista di linee (generate dal comando) da mantenere e quindi le si filtra usando la funzione `filter()` e `any()` (ricordiamo che quest'ultimo restituisce `True` se almeno uno degli eventi generati col codice scritto in essa è `True`).
  Sapendo che ogni riga segue la convenzione `chiave:valore` tramite la funzione `map()` vengono iterate tutte le righe e divise con la funzione `split()`, così da ottenere da ognuna di questa chiave e valore; Questi vengono poi inseriti nella variabile `toreturn` che diventa quindi un dizionario e pasato alla funzione di successo `command_success` nel campo `data`

Return Se le operazioni vanno a buon fine restituisce il dizionario di successo. Per il suo contenuto distinguiamo invece due casi:

- Se l'operazione è *status* il dizionario di successo conterrà nel campo `data` un dizionario contenente alcune informazioni sullo stato attuale del demone di apache
- Se invece si è richiesto qualsiasi altra operazione non ci sono dati da restituire, ma è stato creato un mongolog il cui object id è contenuto nel campo `logid` del dizionario di successo restituito

3.7.3 manageobjs

```
#NOTE: Must not be called directly
def manageobjs(filename, op):

    logid = mongolog( locals() )

    command = [op, filename]

    try:
        #Shrinks the output ---------.
        #                            |
        #                            v
```

```
        check_call(command, stdout=DEVNULL)
    except CalledProcessError as e:
        return command_error( e, command, logid )

    return command_success( logid=logid )

#Call a function with different parameters
def activatevhost(filename): return manageobjs(filename, op='a2ensite')
def deactivatevhost(filename): return manageobjs(filename, op='a2dissite'
    )
def activatemod(filename): return manageobjs(filename, op='a2enmod')
def deactivatemod(filename): return manageobjs(filename, op='a2dismod')
def activateconf(filename): return manageobjs(filename, op='a2enconf')
def deactivateconf(filename): return manageobjs(filename, op='a2disconf')
```

La funzione nasce per gestire gli oggetti di apache, ossia per attivare o disattivare vhosts, mods e conf. Come per le altre funzioni della libreria anche questa non va chiamata direttamente ma tramite le funzioni alias che si vede nelle ultime 6 righe del codice.
ATTENZIONE: Si ricorda che dopo l'attivazione o la disattivazione di un oggetto apache deve essere ricaricato (reload) o riavviato (restart) per attuare le modifiche. Ci pensa il frontend ad informare l'utente e a dargli la possibilità di ricaricare o restartare il demone di apache.

Parametri Non prendiamo in considerazione la funzione principale ma solo i suoi alias. Questi accettano un solo parametro che è il nome dell'oggetto apache da attivare o disattivare e che coincide col nome del file stesso. Viene poi chiamata la funzione principale passandogli anche nel parametro `op` l'operazione da compiere.

Funzionamento Essendo una operazione sensibile (in quanto potrebbe compromettere il buon funzionamento di apache) viene prima creato un documento mongo contenente le specifiche dell'operazione. Poi semplicemente viene costruito il comando e lanciato come si vede nel codice. Viene infine chiamata la funzione di successo `command_success()` passandogli nel campo `logid` l'object id dell'oggetto mongo creato.

Return Viene restituito il dizionario di successo così come generato dalla funzione command_success contenente al campo `logid` l'object id del docuemnto mongo creato contenente le specifiche dell'operazione appena eseguita.

3.7.4 Lettura e scrittura degli oggeti di apache

Sono state omesse nella libreria le funzioni per la modifica degli oggetti di apache in quanto questa può essere effettuata utilizzata la funzione writefile della libreria Sistema, e così viene fatto dal frontend.

3.8 File

La sezione inerente i file (la cui libreria è chiamata *systemfile.py*, a causa della libreria *File* già esistente in python) nasce per la gestione, ricerca ed indicizzazione dei file.

3.8.1 updatedb

```
def updatedb():

    try:
        command = ['updatedb']
        check_call(command, stderr=PIPE)
    except CalledProcessError as e:
        return command_error( e, command )

    return command_success()
```

Questa funzione nasce col presupposto che nel sistema sia presente il pacchetto `mlocate`. Questo pacchetto è stato creato per far fronte al problema della lentezza della ricerca di file nell'intero sistema. Ciò è ottenuto attraverso la creazione di un database indicizzato di tutti i file del sistema che sia in formato binario per velocizzare le operazioni di ricerca.
La funzione quindi nasce per aggiornare questo database utilizzando il frontend di nomodo. Per testare l'utilità di questo applicativo basta confrontare la velocità di ricerca dei file col comando `locate` e la classica `find`.

Parametri La funzione non prende paramtri; tutte le configurazioni vanno inserite nel file `/etc/updatedb.conf`, ma il cambio di configurazione si rende necesario solo per file system speciali come ad esempio l'IBM Spectrun Scale o GPFS; non è quindi necessario eseguire altre operazioni se non installare lo stesso pacchetto `mlocate`.

Funzionamento La funzione non fa altro che lanciare il comando `updatedb` a terminale per aggiornare il database indicizzato dei file.

Return Restituisce il dizionario di successo in cui sia `data` che `logid` sono a `None`. In sintesi non restituisce niente se non un codice di ritorno per sapere se l'operazione è andata a buon fine.

3.8.2 locate

```
def locate(name, insensitive=True):

    #Cannot search on empty string
    if not name:
        return command_error( returncode=255, stderr='Empty search string
            not allowed' )

    try:
        command = ['locate', '-i', name]
        if insensitive is False: command.pop(1)

        found = check_output(command, stderr=PIPE, universal_newlines=
            True).splitlines()
    except CalledProcessError as e:
        return command_error( e, command )

    return command_success( data=found )
```

Questa funzione prende lo stesso nome dell comando che si va a lanciare sul terminale ed è utile alla ricerca di file nel sistema utilizzando il database indicizzato generato ed aggiornato tramite la funzione `updatedb`.

Parametri Prende 2 parametri:

- `name`: è il nome del file da cercare. Il nome può anche essere solo parziale
- `insensitive=True`: indica se la ricerca deve essere effettuata ignorando il case del nome file cercato, cioè se la ricerca non distingue tra caratteri maiuscoli e minuscoli

Funzionamento Viene innanzitutto effettuata una verifica sul valore del parametro `name`, in quanto se l'utente effettua una ricerca su stringa vuota il comando restituisce la lista di tutti i file di sistema. Questo controllo viene anche effettuato dal frontend ma nel mondo dell'informatica la sicurezza non è mai troppa.
Viene quindi vevrificato il parametro `insensitive` e se `False` viene rimosso l'argomento `-i` dal comando da lanciare in quanto l'utente (o il fontend) vuole che la ricerca distingua tra caratteri maiuscoli e caratteri minuscoli.
Viene quindi lanciato il comando `locate` passandogli il nome file che restituisce una lista di risultati. Questa lista viene divisa per righe (`splitlines()`), memorizzata nella variabile `found` e restituita al frontend attraverso la funzione di successo `command_success`.

Return Restituisce il dizionario di successo generato dalla funzione `command_success`, contenente al campo `data` una lista di stringhe dove ogni stringa è il percorso di un file trovato nel sistema e che il cui nome coincida in maniera totale o parziale con la stringa contenuta nella variabile `name`.

3.8.3 Altre funzioni della libreria systemfile

Le funzioni per la scrittura e la rimozione di un file non sono state inserite qui ma direttamente nella libreria Utilities in quanto è più comune includere questa libreria per effettuare tali operazioni piuttosto che la libreria File.

3.9 Logs

La libreria log contiene la funzione che permette di effettuare ricerche tra i mongolog. Come già detto i mongolog sono documenti di mongo che vengono memorizzati da nomodo all'atto di operazioni da parte dello stesso che modificano il sistema in qualche modo e le cui operazioni devono quindi essere memorizzate per tenerne traccia in caso di eventi infausti.
Nella libreria viene usato il database mongo di nomodo le cui credenziali, indirizzo ecc. vengono ricavate dal file *utilities.py* attraverso le variabili `client` e `db` richiamate in questo modo:

```
from utilities import client, db
```

3.9.1

```
def getlog( objectid = None, funname = None, status = None, dategte = "
    1970-01-01", datelte = datetime.datetime.now().strftime("%Y-%m-%d") ):

    query = dict()
    dategte = dateutil.parser.parse( dategte )
```

```
    datelte = dateutil.parser.parse( datelte )

    if objectid:
        query = {"_id" : objectid }
    else:
        query.update({ "date": {"$gte": dategte, "$lte": datelte} })
        if funname: query.update({ "funname" : funname })
        if status: query.update({ "status": status })


    found = list()
    for log in db.log.find( query ):
        found.append( log )

    return command_success( data=found )
```

La funzione nasce, date delle direttive in ingresso, per cercare e restituire i mongolog voluti.

Parametri Accetta 5 parametri:

- `objectid = None`: è l'object id dell mongolog che si vuole. Il passaggio di questo parametro implica che tutti gli altri parametri verranno ignorati in quanto essendo questo un id identifica univocamente uno ed un solo oggetto
- `funname = None`: è in nome della funzione che ha effettuato l'operazione e quindi creato il mongolog che si vuole cercare
- `status = None`: è lo stato dell'operazione e può essere "success" o "error"
- `dategte = "1970-01-01"`: il nome significa "date greater than or equal" ed indica che i mongolog che si devono cercare devono essere stati memorizzati almeno in questa data, quindi nell'intervallo tra `dategte` e la data odierna. Il valore di default è impostato alla linux epoch, ossia alla prima data utile per i sistemi UNIX based
- `datelte = datetime.datetime.now().strftime("%Y-%m-%d")`: il nome significa "date lesser than or equal" ed indica che i mongolog che si devono cercare devono essere stati memorizzati prima di questa data, quindi nell'intervallo tra `dategte` e `datelte`. Il valore di default è impostato alla data odierna ricavata chiamando la funzione datetime.now()

Funzionamento Nella prime tre righe si convertono le date passate come argomento in un formato riconoscibile dalla libreria **pymongo** e quindi adatto ad eseguire query. Si verifica quindi se si stato passato o meno l'object id. Se è stati passato gli altri parametri vengono ignorati e la query viene creata usando solamente questo parametro; viene altrimenti creata la query inserendo sempre le date `dategte` e `datelte` e gli altri parametri invece solo se sono stati passati dall'utente. La quantità di parametri passata è direttamente proporzionale alla finezza della ricerca.
Viene quindi infine lanciata la query con la `db.log.find()` che si vede nel codice e il risultati vengono inseriti nella lista `found` che viene restituita all'utente nel dizionario di successo generato dalla funzione `command_success()`.

Return Restituisce il dizionario di successo contenente al campo `data` una lista contenente tutti i log restituiti dalla ricerca che si è effettuata tramite i parametri dell'utente. Se il parametro `objectid` questa lista conterrà un unico elemento identificato dall'objectid stesso.

4 Frontend

Frontend